

Contents

C# documentation

Get started

Introduction

Types

Program building blocks

Major language areas

Tutorials

Choose your first lesson

Browser based tutorials

Hello world

Numbers in C#

Branches and loops

List collections

Work in your local environment

Set up your environment

Numbers in C#

Branches and loops

List collections

Fundamentals

Program structure

Overview

Main method

Top-level statements

Type system

Overview

Namespaces

Classes

Records

Interfaces

Generics

Anonymous Types

Object Oriented programming

Overview

Objects

Inheritance

Polymorphism

Functional techniques

Pattern matching

Discards

Deconstructing tuples and other types

Exceptions and errors

Overview

Using exceptions

Exception handling

Creating and throwing exceptions

Compiler-generated exceptions

Coding style

Identifier names

C# coding conventions

Tutorials

How to display command-line arguments

Introduction to classes

Object-Oriented C#

Inheritance in C# and .NET

Converting types

Build data-driven algorithms with pattern matching

How to handle an exception using try-catch

How to execute cleanup code using finally

What's new in C#

C# 10

C# 9.0

C# 8.0

Compiler breaking changes

C# Version History

Relationships to .NET library

Version compatibility

Tutorials

- Explore record types

- Explore top-level statements

- Explore patterns in objects

- Safely update interfaces with default interface methods

- Create mixin functionality with default interface methods

- Explore indexes and ranges

- Work with nullable reference types

- Generate and consume asynchronous streams

- Write a custom string interpolation handler

Tutorials

- Explore string interpolation - interactive

- Explore string interpolation - in your environment

- Advanced scenarios for string Interpolation

- Console Application

- REST Client

- Work with LINQ

- Use Attributes

C# concepts

- Nullable reference types

- Nullable reference migrations

- Resolve nullable warnings

- Methods

- Properties

- Indexers

- Iterators

- Delegates & events

Introduction to Delegates

System.Delegate and the delegate keyword

Strongly Typed Delegates

Common Patterns for Delegates

Introduction to events

Standard .NET event patterns

The Updated .NET Event Pattern

Distinguishing Delegates and Events

Language-Integrated Query (LINQ)

Overview of LINQ

Query expression basics

LINQ in C#

Write LINQ queries in C#

Query a collection of objects

Return a query from a method

Store the results of a query in memory

Group query results

Create a nested group

Perform a subquery on a grouping operation

Group results by contiguous keys

Dynamically specify predicate filters at runtime

Perform inner joins

Perform grouped joins

Perform left outer joins

Order the results of a join clause

Join by using composite keys

Perform custom join operations

Handle null values in query expressions

Handle exceptions in query expressions

Write safe, efficient code

Expression trees

Introduction to expression trees

[Expression Trees Explained](#)

[Framework Types Supporting Expression Trees](#)

[Executing Expressions](#)

[Interpreting Expressions](#)

[Building Expressions](#)

[Translating Expressions](#)

[Summary](#)

[Native interoperability](#)

[Versioning](#)

[How-to C# articles](#)

[Article index](#)

[Split strings into substrings](#)

[Concatenate strings](#)

[Search strings](#)

[Modify string contents](#)

[Compare strings](#)

[How to catch a non-CLS exception](#)

[The .NET Compiler Platform SDK \(Roslyn APIs\)](#)

[The .NET Compiler Platform SDK \(Roslyn APIs\) overview](#)

[Understand the compiler API model](#)

[Work with syntax](#)

[Work with semantics](#)

[Work with a workspace](#)

[Explore code with the syntax visualizer](#)

[Source Generators](#)

[Quick starts](#)

[Syntax analysis](#)

[Semantic analysis](#)

[Syntax Transformation](#)

[Tutorials](#)

[Build your first analyzer and code fix](#)

[C# programming guide](#)

[Overview](#)

Programming concepts

Overview

Asynchronous programming

Overview

Asynchronous programming scenarios

Task asynchronous programming model

Async return types

Cancel tasks

Cancel a list of tasks

Cancel tasks after a period of time

Process asynchronous tasks as they complete

Asynchronous file access

Attributes

Overview

Creating Custom Attributes

Accessing Attributes by Using Reflection

How to create a C/C++ union by using attributes

Collections

Covariance and contravariance

Overview

Variance in Generic Interfaces

Create Variant Generic Interfaces

Use Variance in Interfaces for Generic Collections

Variance in Delegates

Use Variance in Delegates

Use Variance for Func and Action Generic Delegates

Expression trees

Overview

How to execute expression trees

How to modify expression trees

How to use expression trees to build dynamic queries

Debugging Expression Trees in Visual Studio

DebugView Syntax

Iterators

Language-Integrated Query (LINQ)

Overview

Getting Started with LINQ in C#

Introduction to LINQ Queries

LINQ and Generic Types

Basic LINQ Query Operations

Data Transformations with LINQ

Type Relationships in LINQ Query Operations

Query Syntax and Method Syntax in LINQ

C# Features That Support LINQ

Walkthrough: Writing Queries in C# (LINQ)

Standard Query Operators Overview

Overview

Query Expression Syntax for Standard Query Operators

Classification of Standard Query Operators by Manner of Execution

Sorting Data

Set Operations

Filtering Data

Quantifier Operations

Projection Operations

Partitioning Data

Join Operations

Grouping Data

Generation Operations

Equality Operations

Element Operations

Converting Data Types

Concatenation Operations

Aggregation Operations

LINQ to Objects

Overview

LINQ and Strings

How to articles

[How to count occurrences of a word in a string \(LINQ\)](#)

[How to query for sentences that contain a specified set of words \(LINQ\)](#)

[How to query for characters in a string \(LINQ\)](#)

[How to combine LINQ queries with regular expressions](#)

[How to find the set difference between two lists \(LINQ\)](#)

[How to sort or filter text data by any word or field \(LINQ\)](#)

[How to reorder the fields of a delimited file \(LINQ\)](#)

[How to combine and compare string collections \(LINQ\)](#)

[How to populate object collections from multiple sources \(LINQ\)](#)

[How to split a file into many files by using groups \(LINQ\)](#)

[How to join content from dissimilar files \(LINQ\)](#)

[How to compute column values in a CSV text file \(LINQ\)](#)

LINQ and Reflection

[How to query an assembly's metadata with Reflection \(LINQ\)](#)

LINQ and File Directories

Overview

[How to query for files with a specified attribute or name](#)

[How to group files by extension \(LINQ\)](#)

[How to query for the total number of bytes in a set of folders \(LINQ\)](#)

[How to compare the contents of two folders \(LINQ\)](#)

[How to query for the largest file or files in a directory tree \(LINQ\)](#)

[How to query for duplicate files in a directory tree \(LINQ\)](#)

[How to query the contents of files in a folder \(LINQ\)](#)

[How to query an ArrayList with LINQ](#)

[How to add custom methods for LINQ queries](#)

LINQ to ADO.NET (Portal Page)

[Enabling a Data Source for LINQ Querying](#)

[Visual Studio IDE and Tools Support for LINQ](#)

Reflection

Serialization (C#)

Overview

How to write object data to an XML file

How to read object data from an XML file

Walkthrough: Persisting an Object in Visual Studio

Statements, expressions, and operators

Overview

Statements

Expression-bodied members

Equality and equality comparisons

Equality comparisons

How to define value equality for a type

How to test for reference equality (identity)

Types

Casting and Type Conversions

Boxing and Unboxing

How to convert a byte array to an int

How to convert a string to a number

How to convert between hexadecimal strings and numeric types

Using Type dynamic

Walkthrough: Creating and Using Dynamic Objects (C# and Visual Basic)

Classes, Structs, and Records

Polymorphism

Versioning with the Override and New Keywords

Knowing When to Use Override and New Keywords

How to override the ToString method

Members

Members overview

Abstract and Sealed Classes and Class Members

Static Classes and Static Class Members

Access Modifiers

Fields

Constants

[How to define abstract properties](#)

[How to define constants in C#](#)

Properties

[Properties overview](#)

[Using Properties](#)

[Interface Properties](#)

[Restricting Accessor Accessibility](#)

[How to declare and use read write properties](#)

[Auto-Implemented Properties](#)

[How to implement a lightweight class with auto-implemented properties](#)

Methods

[Methods overview](#)

[Local functions](#)

[Ref returns and ref locals](#)

[Parameters](#)

[Passing parameters](#)

[Passing Value-Type Parameters](#)

[Passing Reference-Type Parameters](#)

[How to know the difference between passing a struct and passing a class reference to a method](#)

[Implicitly Typed Local Variables](#)

[How to use implicitly typed local variables and arrays in a query expression](#)

[Extension Methods](#)

[How to implement and call a custom extension method](#)

[How to create a new method for an enumeration](#)

[Named and Optional Arguments](#)

[How to use named and optional arguments in Office programming](#)

Constructors

[Constructors overview](#)

[Using Constructors](#)

[Instance Constructors](#)

[Private Constructors](#)

Static Constructors

[How to write a copy constructor](#)

Finalizers

Object and Collection Initializers

[How to initialize objects by using an object initializer](#)

[How to initialize a dictionary with a collection initializer](#)

Nested Types

Partial Classes and Methods

[How to return subsets of element properties in a query](#)

Interfaces

Explicit Interface Implementation

[How to explicitly implement interface members](#)

[How to explicitly implement members of two interfaces](#)

Delegates

Overview

Using Delegates

Delegates with Named vs. Anonymous Methods

[How to combine delegates \(Multicast Delegates\) \(C# Programming Guide\)](#)

[How to declare, instantiate, and use a delegate](#)

Arrays

Overview

Single-Dimensional Arrays

Multidimensional Arrays

Jagged Arrays

Using foreach with Arrays

Passing Arrays as Arguments

Implicitly Typed Arrays

Strings

Programming with strings

[How to determine whether a string represents a numeric value](#)

Indexers

Overview

Using Indexers

Indexers in Interfaces

Comparison Between Properties and Indexers

Events

Overview

How to subscribe to and unsubscribe from events

How to publish events that conform to .NET Guidelines

How to raise base class events in derived classes

How to implement interface events

How to implement custom event accessors

Generics

Generic Type Parameters

Constraints on Type Parameters

Generic Classes

Generic Interfaces

Generic Methods

Generics and Arrays

Generic Delegates

Differences Between C++ Templates and C# Generics

Generics in the Run Time

Generics and Reflection

Generics and Attributes

File System and the Registry

Overview

How to iterate through a directory tree

How to get information about files, folders, and drives

How to create a file or folder

How to copy, delete, and move files and folders

How to provide a progress dialog box for file operations

How to write to a text file

How to read From a text file

How to read a text file one line at a time

[How to create a key in the registry](#)

[Interoperability](#)

[.NET Interoperability](#)

[Interoperability Overview](#)

[How to access Office interop objects by using C# features](#)

[How to use indexed properties in COM interop programming](#)

[How to use platform invoke to play a WAV file](#)

[Walkthrough: Office Programming \(C# and Visual Basic\)](#)

[Example COM Class](#)

[Language reference](#)

[Overview](#)

[Configure language version](#)

[Types](#)

[Value types](#)

[Overview](#)

[Integral numeric types](#)

[nint and nuint native integer types](#)

[Floating-point numeric types](#)

[Built-in numeric conversions](#)

[bool](#)

[char](#)

[Enumeration types](#)

[Structure types](#)

[Tuple types](#)

[Nullable value types](#)

[Reference types](#)

[Features of reference types](#)

[Built-in reference types](#)

[record](#)

[class](#)

[interface](#)

[Nullable reference types](#)

[void](#)

[var](#)

[Built-in types](#)

[Unmanaged types](#)

[Default values](#)

[Keywords](#)

[Overview](#)

[Modifiers](#)

[Access Modifiers](#)

[Quick reference](#)

[Accessibility Levels](#)

[Accessibility Domain](#)

[Restrictions on Using Accessibility Levels](#)

[internal](#)

[private](#)

[protected](#)

[public](#)

[protected internal](#)

[private protected](#)

[abstract](#)

[async](#)

[const](#)

[event](#)

[extern](#)

[in \(generic modifier\)](#)

[new \(member modifier\)](#)

[out \(generic modifier\)](#)

[override](#)

[readonly](#)

[sealed](#)

[static](#)

[unsafe](#)

virtual

volatile

Statement Keywords

Statement categories

Exception Handling Statements

throw

try-catch

try-finally

try-catch-finally

Checked and Unchecked

Overview

checked

unchecked

fixed Statement

Method Parameters

Passing parameters

params

in (Parameter Modifier)

ref

out (Parameter Modifier)

Namespace Keywords

namespace

using

Contexts for using

using Directive

using Statement

extern alias

Generic Type Constraint Keywords

new constraint

where

Access Keywords

base

this

Literal Keywords

null

true and false

default

Contextual Keywords

Quick reference

add

get

init

partial (Type)

partial (Method)

remove

set

when (filter condition)

value

yield

Query Keywords

Quick reference

from clause

where clause

select clause

group clause

into

orderby clause

join clause

let clause

ascending

descending

on

equals

by

in

Operators and expressions

Overview

Arithmetic operators

Boolean logical operators

Bitwise and shift operators

Equality operators

Comparison operators

Member access operators and expressions

Type-testing operators and cast expression

User-defined conversion operators

Pointer-related operators

Assignment operators

Lambda expressions

Patterns

+ and += operators

- and -= operators

?: operator

! (null-forgiving) operator

?? and ??= operators

=> operator

:: operator

await operator

default value expressions

delegate operator

is operator

nameof expression

new operator

sizeof operator

stackalloc expression

switch expression

true and false operators

with expression

Operator overloading

Statements

Iteration statements

Selection statements

Jump statements

lock statement

Special characters

Overview

\$ -- string interpolation

@ -- verbatim identifier

Attributes read by the compiler

Global attributes

Caller information

Nullable static analysis

Miscellaneous

Unsafe code and pointers

Preprocessor directives

Compiler options

Overview

Language options

Output options

Input options

Error and warning options

Code generation options

Security options

Resources options

Miscellaneous options

Advanced options

XML documentation comments

Generate API documentation

Recommended tags

Examples

Compiler messages

Specifications

C# 6.0 draft specification

Introduction

Lexical structure

Basic concepts

Types

Variables

Conversions

Expressions

Statements

Namespaces

Classes

Structs

Arrays

Interfaces

Enums

Delegates

Exceptions

Attributes

Unsafe code

Documentation comments

C# 7.0 - 10.0 features

C# 7.0 features

Pattern matching

Local functions

Out variable declarations

Throw expressions

Binary literals

Digit separators

Async task types

C# 7.1 features

- Async main method
- Default expressions
- Infer tuple names
- Pattern matching with generics

C# 7.2 features

- Readonly references
- Compile-time safety for ref-like types
- Non-trailing named arguments
- Private protected
- Conditional ref
- Leading digit separator

C# 7.3 features

- Unmanaged generic type constraints
- Indexing ``fixed`` fields should not require pinning regardless of the movable/unmovable context
- Pattern-based ``fixed`` statement
- Ref local reassignment
- Stackalloc array initializers
- Auto-implemented property field-targeted attributes
- Expression variables in initializers
- Tuple equality (`==`) and inequality (`!=`)
- Improved overload candidates

C# 8.0 features

- Nullable reference types - proposal
- Recursive pattern matching
- Default interface methods
- Async streams
- Ranges
- Pattern based using and using declarations
- Static local functions
- Null coalescing assignment
- Readonly instance members

Nested stackalloc

C# 9.0 features

Records

Top-level statements

Nullable reference types - specification

Pattern matching enhancements

Init only setters

Target-typed new expressions

Module initializers

Extending partial methods

Static anonymous functions

Target-typed conditional expression

Covariant return types

Extension GetEnumerator in foreach loops

Lambda discard parameters

Attributes on local functions

Native sized integers

Function pointers

Suppress emitting localsinit flag

Unconstrained type parameter annotations

C# 10 features

Record structs

Parameterless struct constructors

Global using directive

File scoped namespaces

Extended property patterns

Improved interpolated strings

Constant interpolated strings

Lambda improvements

Caller argument expression

Enhanced #line directives

Generic attributes

Improved definite assignment analysis

AsyncMethodBuilder override

A tour of the C# language

12/28/2021 • 13 minutes to read • [Edit Online](#)

C# (pronounced "See Sharp") is a modern, object-oriented, and type-safe programming language. C# enables developers to build many types of secure and robust applications that run in .NET. C# has its roots in the C family of languages and will be immediately familiar to C, C++, Java, and JavaScript programmers. This tour provides an overview of the major components of the language in C# 8 and earlier. If you want to explore the language through interactive examples, try the [introduction to C#](#) tutorials.

C# is an object-oriented, **component-oriented** programming language. C# provides language constructs to directly support these concepts, making C# a natural language in which to create and use software components. Since its origin, C# has added features to support new workloads and emerging software design practices. At its core, C# is an **object-oriented** language. You define types and their behavior.

Several C# features help create robust and durable applications. **Garbage collection** automatically reclaims memory occupied by unreachable unused objects. **Nullable types** guard against variables that don't refer to allocated objects. **Exception handling** provides a structured and extensible approach to error detection and recovery. **Lambda expressions** support functional programming techniques. **Language Integrated Query (LINQ)** syntax creates a common pattern for working with data from any source. Language support for **asynchronous operations** provides syntax for building distributed systems. C# has a **unified type system**. All C# types, including primitive types such as `int` and `double`, inherit from a single root `object` type. All types share a set of common operations. Values of any type can be stored, transported, and operated upon in a consistent manner. Furthermore, C# supports both user-defined **reference types** and **value types**. C# allows dynamic allocation of objects and in-line storage of lightweight structures. C# supports generic methods and types, which provide increased type safety and performance. C# provides iterators, which enable implementers of collection classes to define custom behaviors for client code.

C# emphasizes **versioning** to ensure programs and libraries can evolve over time in a compatible manner. Aspects of C#'s design that were directly influenced by versioning considerations include the separate `virtual` and `override` modifiers, the rules for method overload resolution, and support for explicit interface member declarations.

.NET architecture

C# programs run on .NET, a virtual execution system called the common language runtime (CLR) and a set of class libraries. The CLR is the implementation by Microsoft of the common language infrastructure (CLI), an international standard. The CLI is the basis for creating execution and development environments in which languages and libraries work together seamlessly.

Source code written in C# is compiled into an **intermediate language (IL)** that conforms to the CLI specification. The IL code and resources, such as bitmaps and strings, are stored in an assembly, typically with an extension of `.dll`. An assembly contains a manifest that provides information about the assembly's types, version, and culture.

When the C# program is executed, the assembly is loaded into the CLR. The CLR performs Just-In-Time (JIT) compilation to convert the IL code to native machine instructions. The CLR provides other services related to automatic garbage collection, exception handling, and resource management. Code that's executed by the CLR is sometimes referred to as "managed code." "Unmanaged code," is compiled into native machine language that targets a specific platform.

Language interoperability is a key feature of .NET. IL code produced by the C# compiler conforms to the Common Type Specification (CTS). IL code generated from C# can interact with code that was generated from

the .NET versions of F#, Visual Basic, C++. There are more than 20 other CTS-compliant languages. A single assembly may contain multiple modules written in different .NET languages. The types can reference each other as if they were written in the same language.

In addition to the run time services, .NET also includes extensive libraries. These libraries support many different workloads. They're organized into namespaces that provide a wide variety of useful functionality. The libraries include everything from file input and output to string manipulation to XML parsing, to web application frameworks to Windows Forms controls. The typical C# application uses the .NET class library extensively to handle common "plumbing" chores.

For more information about .NET, see [Overview of .NET](#).

Hello world

The "Hello, World" program is traditionally used to introduce a programming language. Here it is in C#:

```
using System;

class Hello
{
    static void Main()
    {
        Console.WriteLine("Hello, World");
    }
}
```

The "Hello, World" program starts with a `using` directive that references the `System` namespace. Namespaces provide a hierarchical means of organizing C# programs and libraries. Namespaces contain types and other namespaces—for example, the `System` namespace contains a number of types, such as the `Console` class referenced in the program, and a number of other namespaces, such as `IO` and `Collections`. A `using` directive that references a given namespace enables unqualified use of the types that are members of that namespace. Because of the `using` directive, the program can use `Console.WriteLine` as shorthand for `System.Console.WriteLine`.

The `Hello` class declared by the "Hello, World" program has a single member, the method named `Main`. The `Main` method is declared with the `static` modifier. While instance methods can reference a particular enclosing object instance using the keyword `this`, static methods operate without reference to a particular object. By convention, a static method named `Main` serves as the entry point of a C# program.

The output of the program is produced by the `WriteLine` method of the `Console` class in the `System` namespace. This class is provided by the standard class libraries, which, by default, are automatically referenced by the compiler.

Types and variables

A *type* defines the structure and behavior of any data in C#. The declaration of a type may include its members, base type, interfaces it implements, and operations permitted for that type. A *variable* is a label that refers to an instance of a specific type.

There are two kinds of types in C#: *value types* and *reference types*. Variables of value types directly contain their data. Variables of reference types store references to their data, the latter being known as objects. With reference types, it's possible for two variables to reference the same object and possible for operations on one variable to affect the object referenced by the other variable. With value types, the variables each have their own copy of the data, and it isn't possible for operations on one to affect the other (except for `ref` and `out` parameter variables).

An *identifier* is a variable name. An identifier is a sequence of unicode characters without any whitespace. An identifier may be a C# reserved word, if it's prefixed by `@`. Using a reserved word as an identifier can be useful when interacting with other languages.

C#'s value types are further divided into *simple types*, *enum types*, *struct types*, *nullable value types*, and *tuple value types*. C#'s reference types are further divided into *class types*, *interface types*, *array types*, and *delegate types*.

The following outline provides an overview of C#'s type system.

- **Value types**
 - **Simple types**
 - **Signed integral:** `sbyte`, `short`, `int`, `long`
 - **Unsigned integral:** `byte`, `ushort`, `uint`, `ulong`
 - **Unicode characters:** `char`, which represents a UTF-16 code unit
 - **IEEE binary floating-point:** `float`, `double`
 - **High-precision decimal floating-point:** `decimal`
 - **Boolean:** `bool`, which represents Boolean values—values that are either `true` or `false`
 - **Enum types**
 - User-defined types of the form `enum E {...}`. An `enum` type is a distinct type with named constants. Every `enum` type has an underlying type, which must be one of the eight integral types. The set of values of an `enum` type is the same as the set of values of the underlying type.
 - **Struct types**
 - User-defined types of the form `struct S {...}`
 - **Nullable value types**
 - Extensions of all other value types with a `null` value
 - **Tuple value types**
 - User-defined types of the form `(T1, T2, ...)`
- **Reference types**
 - **Class types**
 - Ultimate base class of all other types: `object`
 - **Unicode strings:** `string`, which represents a sequence of UTF-16 code units
 - User-defined types of the form `class C {...}`
 - **Interface types**
 - User-defined types of the form `interface I {...}`
 - **Array types**
 - Single-dimensional, multi-dimensional, and jagged. For example: `int[]`, `int[,]`, and `int[][]`
 - **Delegate types**
 - User-defined types of the form `delegate int D(...)`

C# programs use *type declarations* to create new types. A type declaration specifies the name and the members of the new type. Six of C#'s categories of types are user-definable: class types, struct types, interface types, enum types, delegate types, and tuple value types. You can also declare `record` types, either `record struct`, or `record class`. Record types have compiler-synthesized members. You use records primarily for storing values, with minimal associated behavior.

- A `class` type defines a data structure that contains data members (fields) and function members (methods, properties, and others). Class types support single inheritance and polymorphism, mechanisms whereby derived classes can extend and specialize base classes.
- A `struct` type is similar to a class type in that it represents a structure with data members and function

members. However, unlike classes, structs are value types and don't typically require heap allocation. Struct types don't support user-specified inheritance, and all struct types implicitly inherit from type `object`.

- An `interface` type defines a contract as a named set of public members. A `class` or `struct` that implements an `interface` must provide implementations of the interface's members. An `interface` may inherit from multiple base interfaces, and a `class` or `struct` may implement multiple interfaces.
- A `delegate` type represents references to methods with a particular parameter list and return type. Delegates make it possible to treat methods as entities that can be assigned to variables and passed as parameters. Delegates are analogous to function types provided by functional languages. They're also similar to the concept of function pointers found in some other languages. Unlike function pointers, delegates are object-oriented and type-safe.

The `class`, `struct`, `interface`, and `delegate` types all support generics, whereby they can be parameterized with other types.

C# supports single-dimensional and multi-dimensional arrays of any type. Unlike the types listed above, array types don't have to be declared before they can be used. Instead, array types are constructed by following a type name with square brackets. For example, `int[]` is a single-dimensional array of `int`, `int[,]` is a two-dimensional array of `int`, and `int[][]` is a single-dimensional array of single-dimensional arrays, or a "jagged" array, of `int`.

Nullable types don't require a separate definition. For each non-nullable type `T`, there's a corresponding nullable type `T?`, which can hold an additional value, `null`. For instance, `int?` is a type that can hold any 32-bit integer or the value `null`, and `string?` is a type that can hold any `string` or the value `null`.

C#'s type system is unified such that a value of any type can be treated as an `object`. Every type in C# directly or indirectly derives from the `object` class type, and `object` is the ultimate base class of all types. Values of reference types are treated as objects simply by viewing the values as type `object`. Values of value types are treated as objects by performing *boxing* and *unboxing operations*. In the following example, an `int` value is converted to `object` and back again to `int`.

```
int i = 123;
object o = i;    // Boxing
int j = (int)o;  // Unboxing
```

When a value of a value type is assigned to an `object` reference, a "box" is allocated to hold the value. That box is an instance of a reference type, and the value is copied into that box. Conversely, when an `object` reference is cast to a value type, a check is made that the referenced `object` is a box of the correct value type. If the check succeeds, the value in the box is copied to the value type.

C#'s unified type system effectively means that value types are treated as `object` references "on demand." Because of the unification, general-purpose libraries that use type `object` can be used with all types that derive from `object`, including both reference types and value types.

There are several kinds of *variables* in C#, including fields, array elements, local variables, and parameters. Variables represent storage locations. Every variable has a type that determines what values can be stored in the variable, as shown below.

- Non-nullable value type
 - A value of that exact type
- Nullable value type
 - A `null` value or a value of that exact type
- `object`
 - A `null` reference, a reference to an object of any reference type, or a reference to a boxed value of

any value type

- Class type
 - A `null` reference, a reference to an instance of that class type, or a reference to an instance of a class derived from that class type
- Interface type
 - A `null` reference, a reference to an instance of a class type that implements that interface type, or a reference to a boxed value of a value type that implements that interface type
- Array type
 - A `null` reference, a reference to an instance of that array type, or a reference to an instance of a compatible array type
- Delegate type
 - A `null` reference or a reference to an instance of a compatible delegate type

Program structure

The key organizational concepts in C# are *programs*, *namespaces*, *types*, *members*, and *assemblies*. Programs declare types, which contain members and can be organized into namespaces. Classes, structs, and interfaces are examples of types. Fields, methods, properties, and events are examples of members. When C# programs are compiled, they're physically packaged into assemblies. Assemblies typically have the file extension `.exe` or `.dll`, depending on whether they implement *applications* or *libraries*, respectively.

As a small example, consider an assembly that contains the following code:

```
namespace Acme.Collections;

public class Stack<T>
{
    Entry _top;

    public void Push(T data)
    {
        _top = new Entry(_top, data);
    }

    public T Pop()
    {
        if (_top == null)
        {
            throw new InvalidOperationException();
        }
        T result = _top.Data;
        _top = _top.Next;

        return result;
    }

    class Entry
    {
        public Entry Next { get; set; }
        public T Data { get; set; }

        public Entry(Entry next, T data)
        {
            Next = next;
            Data = data;
        }
    }
}
```

The fully qualified name of this class is `Acme.Collections.Stack`. The class contains several members: a field named `_top`, two methods named `Push` and `Pop`, and a nested class named `Entry`. The `Entry` class further contains three members: a property named `Next`, a property named `Data`, and a constructor. The `Stack` is a *generic* class. It has one type parameter, `T` that is replaced with a concrete type when it's used.

A *stack* is a "first in - last out" (FILO) collection. New elements are added to the top of the stack. When an element is removed, it's removed from the top of the stack. The previous example declares the `Stack` type that defines the storage and behavior for a stack. You can declare a variable that refers to an instance of the `Stack` type to use that functionality.

Assemblies contain executable code in the form of Intermediate Language (IL) instructions, and symbolic information in the form of metadata. Before it's executed, the Just-In-Time (JIT) compiler of .NET Common Language Runtime converts the IL code in an assembly to processor-specific code.

Because an assembly is a self-describing unit of functionality containing both code and metadata, there's no need for `#include` directives and header files in C#. The public types and members contained in a particular assembly are made available in a C# program simply by referencing that assembly when compiling the program. For example, this program uses the `Acme.Collections.Stack` class from the `acme.dll` assembly:

```
class Example
{
    public static void Main()
    {
        var s = new Acme.Collections.Stack<int>();
        s.Push(1); // stack contains 1
        s.Push(10); // stack contains 1, 10
        s.Push(100); // stack contains 1, 10, 100
        Console.WriteLine(s.Pop()); // stack contains 1, 10
        Console.WriteLine(s.Pop()); // stack contains 1
        Console.WriteLine(s.Pop()); // stack is empty
    }
}
```

To compile this program, you would need to *reference* the assembly containing the stack class defined in the earlier example.

C# programs can be stored in several source files. When a C# program is compiled, all of the source files are processed together, and the source files can freely reference each other. Conceptually, it's as if all the source files were concatenated into one large file before being processed. Forward declarations are never needed in C# because, with few exceptions, declaration order is insignificant. C# doesn't limit a source file to declaring only one public type nor does it require the name of the source file to match a type declared in the source file.

Further articles in this tour explain these organizational blocks.

NEXT

Types and members

12/28/2021 • 6 minutes to read • [Edit Online](#)

As an object-oriented language, C# supports the concepts of encapsulation, inheritance, and polymorphism. A class may inherit directly from one parent class, and it may implement any number of interfaces. Methods that override virtual methods in a parent class require the `override` keyword as a way to avoid accidental redefinition. In C#, a struct is like a lightweight class; it's a stack-allocated type that can implement interfaces but doesn't support inheritance. C# provides `record class`, and `record struct` types which are types whose purpose is primarily storing data values.

Classes and objects

Classes are the most fundamental of C#'s types. A class is a data structure that combines state (fields) and actions (methods and other function members) in a single unit. A class provides a definition for *instances* of the class, also known as *objects*. Classes support *inheritance* and *polymorphism*, mechanisms whereby *derived classes* can extend and specialize *base classes*.

New classes are created using class declarations. A class declaration starts with a header. The header specifies:

- The attributes and modifiers of the class
- The name of the class
- The base class (when inheriting from a [base class](#))
- The interfaces implemented by the class.

The header is followed by the class body, which consists of a list of member declarations written between the delimiters `{` and `}`.

The following code shows a declaration of a simple class named `Point`:

```
public class Point
{
    public int X { get; }
    public int Y { get; }

    public Point(int x, int y) => (X, Y) = (x, y);
}
```

Instances of classes are created using the `new` operator, which allocates memory for a new instance, invokes a constructor to initialize the instance, and returns a reference to the instance. The following statements create two `Point` objects and store references to those objects in two variables:

```
var p1 = new Point(0, 0);
var p2 = new Point(10, 20);
```

The memory occupied by an object is automatically reclaimed when the object is no longer reachable. It's not necessary or possible to explicitly deallocate objects in C#.

Type parameters

Generic classes define [type parameters](#). Type parameters are a list of type parameter names enclosed in angle brackets. Type parameters follow the class name. The type parameters can then be used in the body of the class declarations to define the members of the class. In the following example, the type parameters of `Pair` are

TFirst and TSecond :

```
public class Pair<TFirst, TSecond>
{
    public TFirst First { get; }
    public TSecond Second { get; }

    public Pair(TFirst first, TSecond second) =>
        (First, Second) = (first, second);
}
```

A class type that is declared to take type parameters is called a *generic class type*. Struct, interface, and delegate types can also be generic. When the generic class is used, type arguments must be provided for each of the type parameters:

```
var pair = new Pair<int, string>(1, "two");
int i = pair.First;    //TFirst int
string s = pair.Second; //TSecond string
```

A generic type with type arguments provided, like `Pair<int, string>` above, is called a *constructed type*.

Base classes

A class declaration may specify a base class. Follow the class name and type parameters with a colon and the name of the base class. Omitting a base class specification is the same as deriving from type `object`. In the following example, the base class of `Point3D` is `Point`. From the first example, the base class of `Point` is `object`:

```
public class Point3D : Point
{
    public int Z { get; set; }

    public Point3D(int x, int y, int z) : base(x, y)
    {
        Z = z;
    }
}
```

A class inherits the members of its base class. Inheritance means that a class implicitly contains almost all members of its base class. A class doesn't inherit the instance and static constructors, and the finalizer. A derived class can add new members to those members it inherits, but it can't remove the definition of an inherited member. In the previous example, `Point3D` inherits the `x` and `y` members from `Point`, and every `Point3D` instance contains three properties, `x`, `y`, and `z`.

An implicit conversion exists from a class type to any of its base class types. A variable of a class type can reference an instance of that class or an instance of any derived class. For example, given the previous class declarations, a variable of type `Point` can reference either a `Point` or a `Point3D`:

```
Point a = new(10, 20);
Point b = new Point3D(10, 20, 30);
```

Structs

Classes define types that support inheritance and polymorphism. They enable you to create sophisticated behaviors based on hierarchies of derived classes. By contrast, *struct* types are simpler types whose primary purpose is to store data values. Structs can't declare a base type; they implicitly derive from [System.ValueType](#).

You can't derive other `struct` types from a `struct` type. They're implicitly sealed.

```
public struct Point
{
    public double X { get; }
    public double Y { get; }

    public Point(double x, double y) => (X, Y) = (x, y);
}
```

Interfaces

An *interface* defines a contract that can be implemented by classes and structs. You define an *interface* to declare capabilities that are shared among distinct types. For example, the [System.Collections.Generic.IEnumerable<T>](#) interface defines a consistent way to traverse all the items in a collection, such as an array. An interface can contain methods, properties, events, and indexers. An interface typically doesn't provide implementations of the members it defines—it merely specifies the members that must be supplied by classes or structs that implement the interface.

Interfaces may employ *multiple inheritance*. In the following example, the interface `IComboBox` inherits from both `ITextBox` and `IListBox`.

```
interface IControl
{
    void Paint();
}

interface ITextBox : IControl
{
    void SetText(string text);
}

interface IListBox : IControl
{
    void SetItems(string[] items);
}

interface IComboBox : ITextBox, IListBox { }
```

Classes and structs can implement multiple interfaces. In the following example, the class `EditBox` implements both `IControl` and `IDataBound`.

```
interface IDataBound
{
    void Bind(Binder b);
}

public class EditBox : IControl, IDataBound
{
    public void Paint() { }
    public void Bind(Binder b) { }
}
```

When a class or struct implements a particular interface, instances of that class or struct can be implicitly converted to that interface type. For example

```
EditText editBox = new();
IControl control = editBox;
IDataBound dataBound = editBox;
```

Enums

An *Enum* type defines a set of constant values. The following `enum` declares constants that define different root vegetables:

```
public enum SomeRootVegetable
{
    HorseRadish,
    Radish,
    Turnip
}
```

You can also define an `enum` to be used in combination as flags. The following declaration declares a set of flags for the four seasons. Any combination of the seasons may be applied, including an `All` value that includes all seasons:

```
[Flags]
public enum Seasons
{
    None = 0,
    Summer = 1,
    Autumn = 2,
    Winter = 4,
    Spring = 8,
    All = Summer | Autumn | Winter | Spring
}
```

The following example shows declarations of both the preceding enums:

```
var turnip = SomeRootVegetable.Turnip;

var spring = Seasons.Spring;
var startingOnEquinox = Seasons.Spring | Seasons.Autumn;
var theYear = Seasons.All;
```

Nullable types

Variables of any type may be declared as *non-nullable* or *nullable*. A nullable variable can hold an additional `null` value, indicating no value. Nullable Value types (structs or enums) are represented by `System.Nullable<T>`. Non-nullable and Nullable Reference types are both represented by the underlying reference type. The distinction is represented by metadata read by the compiler and some libraries. The compiler provides warnings when nullable references are dereferenced without first checking their value against `null`. The compiler also provides warnings when non-nullable references are assigned a value that may be `null`. The following example declares a *nullable int*, initializing it to `null`. Then, it sets the value to `5`. It demonstrates the same concept with a *nullable string*. For more information, see [nullable value types](#) and [nullable reference types](#).

```
int? optionalInt = default;  
optionalInt = 5;  
string? optionalText = default;  
optionalText = "Hello World.";
```

Tuples

C# supports *tuples*, which provides concise syntax to group multiple data elements in a lightweight data structure. You instantiate a tuple by declaring the types and names of the members between `(` and `)`, as shown in the following example:

```
(double Sum, int Count) t2 = (4.5, 3);  
Console.WriteLine($"Sum of {t2.Count} elements is {t2.Sum}.");  
//Output:  
//Sum of 3 elements is 4.5.
```

Tuples provide an alternative for data structure with multiple members, without using the building blocks described in the next article.

[PREVIOUS](#)[NEXT](#)

Program building blocks

12/28/2021 • 23 minutes to read • [Edit Online](#)

The types described in the previous article are built using these building blocks: *members*, *expressions*, and *statements*.

Members

The members of a `class` are either *static members* or *instance members*. Static members belong to classes, and instance members belong to objects (instances of classes).

The following list provides an overview of the kinds of members a class can contain.

- **Constants:** Constant values associated with the class
- **Fields:** Variables that are associated with the class
- **Methods:** Actions that can be performed by the class
- **Properties:** Actions associated with reading and writing named properties of the class
- **Indexers:** Actions associated with indexing instances of the class like an array
- **Events:** Notifications that can be generated by the class
- **Operators:** Conversions and expression operators supported by the class
- **Constructors:** Actions required to initialize instances of the class or the class itself
- **Finalizers:** Actions done before instances of the class are permanently discarded
- **Types:** Nested types declared by the class

Accessibility

Each member of a class has an associated accessibility, which controls the regions of program text that can access the member. There are six possible forms of accessibility. The access modifiers are summarized below.

- `public`: Access isn't limited.
- `private`: Access is limited to this class.
- `protected`: Access is limited to this class or classes derived from this class.
- `internal`: Access is limited to the current assembly (`.exe` or `.dll`).
- `protected internal`: Access is limited to this class, classes derived from this class, or classes within the same assembly.
- `private protected`: Access is limited to this class or classes derived from this type within the same assembly.

Fields

A *field* is a variable that is associated with a class or with an instance of a class.

A field declared with the static modifier defines a static field. A static field identifies exactly one storage location. No matter how many instances of a class are created, there's only ever one copy of a static field.

A field declared without the static modifier defines an instance field. Every instance of a class contains a separate copy of all the instance fields of that class.

In the following example, each instance of the `Color` class has a separate copy of the `R`, `G`, and `B` instance fields, but there's only one copy of the `Black`, `White`, `Red`, `Green`, and `Blue` static fields:

```

public class Color
{
    public static readonly Color Black = new(0, 0, 0);
    public static readonly Color White = new(255, 255, 255);
    public static readonly Color Red = new(255, 0, 0);
    public static readonly Color Green = new(0, 255, 0);
    public static readonly Color Blue = new(0, 0, 255);

    public byte R;
    public byte G;
    public byte B;

    public Color(byte r, byte g, byte b)
    {
        R = r;
        G = g;
        B = b;
    }
}

```

As shown in the previous example, *read-only fields* may be declared with a `readonly` modifier. Assignment to a read-only field can only occur as part of the field's declaration or in a constructor in the same class.

Methods

A *method* is a member that implements a computation or action that can be performed by an object or class. *Static methods* are accessed through the class. *Instance methods* are accessed through instances of the class.

Methods may have a list of *parameters*, which represent values or variable references passed to the method. Methods have a *return type*, which specifies the type of the value computed and returned by the method. A method's return type is `void` if it doesn't return a value.

Like types, methods may also have a set of type parameters, for which type arguments must be specified when the method is called. Unlike types, the type arguments can often be inferred from the arguments of a method call and need not be explicitly given.

The *signature* of a method must be unique in the class in which the method is declared. The signature of a method consists of the name of the method, the number of type parameters, and the number, modifiers, and types of its parameters. The signature of a method doesn't include the return type.

When a method body is a single expression, the method can be defined using a compact expression format, as shown in the following example:

```

public override string ToString() => "This is an object";

```

Parameters

Parameters are used to pass values or variable references to methods. The parameters of a method get their actual values from the *arguments* that are specified when the method is invoked. There are four kinds of parameters: value parameters, reference parameters, output parameters, and parameter arrays.

A *value parameter* is used for passing input arguments. A value parameter corresponds to a local variable that gets its initial value from the argument that was passed for the parameter. Modifications to a value parameter don't affect the argument that was passed for the parameter.

Value parameters can be optional, by specifying a default value so that corresponding arguments can be omitted.

A *reference parameter* is used for passing arguments by reference. The argument passed for a reference

parameter must be a variable with a definite value. During execution of the method, the reference parameter represents the same storage location as the argument variable. A reference parameter is declared with the `ref` modifier. The following example shows the use of `ref` parameters.

```
static void Swap(ref int x, ref int y)
{
    int temp = x;
    x = y;
    y = temp;
}

public static void SwapExample()
{
    int i = 1, j = 2;
    Swap(ref i, ref j);
    Console.WriteLine($"{i} {j}");    // "2 1"
}
```

An *output parameter* is used for passing arguments by reference. It's similar to a reference parameter, except that it doesn't require that you explicitly assign a value to the caller-provided argument. An output parameter is declared with the `out` modifier. The following example shows the use of `out` parameters using the syntax introduced in C# 7.

```
static void Divide(int x, int y, out int result, out int remainder)
{
    result = x / y;
    remainder = x % y;
}

public static void OutUsage()
{
    Divide(10, 3, out int res, out int rem);
    Console.WriteLine($"{res} {rem}"); // "3 1"
}
```

A *parameter array* permits a variable number of arguments to be passed to a method. A parameter array is declared with the `params` modifier. Only the last parameter of a method can be a parameter array, and the type of a parameter array must be a single-dimensional array type. The `Write` and `WriteLine` methods of the [System.Console](#) class are good examples of parameter array usage. They're declared as follows.

```
public class Console
{
    public static void Write(string fmt, params object[] args) { }
    public static void WriteLine(string fmt, params object[] args) { }
    // ...
}
```

Within a method that uses a parameter array, the parameter array behaves exactly like a regular parameter of an array type. However, in an invocation of a method with a parameter array, it's possible to pass either a single argument of the parameter array type or any number of arguments of the element type of the parameter array. In the latter case, an array instance is automatically created and initialized with the given arguments. This example


```
int x, y, z;
x = 3;
y = 4;
z = 5;
Console.WriteLine("x={0} y={1} z={2}", x, y, z);
```

is equivalent to writing the following.

```
int x = 3, y = 4, z = 5;

string s = "x={0} y={1} z={2}";
object[] args = new object[3];
args[0] = x;
args[1] = y;
args[2] = z;
Console.WriteLine(s, args);
```

Method body and local variables

A method's body specifies the statements to execute when the method is invoked.

A method body can declare variables that are specific to the invocation of the method. Such variables are called *local variables*. A local variable declaration specifies a type name, a variable name, and possibly an initial value. The following example declares a local variable `i` with an initial value of zero and a local variable `j` with no initial value.

```
class Squares
{
    public static void WriteSquares()
    {
        int i = 0;
        int j;
        while (i < 10)
        {
            j = i * i;
            Console.WriteLine($"{i} x {i} = {j}");
            i++;
        }
    }
}
```

C# requires a local variable to be *definitely assigned* before its value can be obtained. For example, if the declaration of the previous `i` didn't include an initial value, the compiler would report an error for the later usages of `i` because `i` wouldn't be definitely assigned at those points in the program.

A method can use `return` statements to return control to its caller. In a method returning `void`, `return` statements can't specify an expression. In a method returning non-void, `return` statements must include an expression that computes the return value.

Static and instance methods

A method declared with a `static` modifier is a *static method*. A static method doesn't operate on a specific instance and can only directly access static members.

A method declared without a `static` modifier is an *instance method*. An instance method operates on a specific instance and can access both static and instance members. The instance on which an instance method was invoked can be explicitly accessed as `this`. It's an error to refer to `this` in a static method.

The following `Entity` class has both static and instance members.

```

class Entity
{
    static int s_nextSerialNo;
    int _serialNo;

    public Entity()
    {
        _serialNo = s_nextSerialNo++;
    }

    public int GetSerialNo()
    {
        return _serialNo;
    }

    public static int GetNextSerialNo()
    {
        return s_nextSerialNo;
    }

    public static void SetNextSerialNo(int value)
    {
        s_nextSerialNo = value;
    }
}

```

Each `Entity` instance contains a serial number (and presumably some other information that isn't shown here). The `Entity` constructor (which is like an instance method) initializes the new instance with the next available serial number. Because the constructor is an instance member, it's permitted to access both the `_serialNo` instance field and the `s_nextSerialNo` static field.

The `GetNextSerialNo` and `SetNextSerialNo` static methods can access the `s_nextSerialNo` static field, but it would be an error for them to directly access the `_serialNo` instance field.

The following example shows the use of the `Entity` class.

```

Entity.SetNextSerialNo(1000);
Entity e1 = new();
Entity e2 = new();
Console.WriteLine(e1.GetSerialNo());           // Outputs "1000"
Console.WriteLine(e2.GetSerialNo());           // Outputs "1001"
Console.WriteLine(Entity.GetNextSerialNo());   // Outputs "1002"

```

The `SetNextSerialNo` and `GetNextSerialNo` static methods are invoked on the class whereas the `GetSerialNo` instance method is invoked on instances of the class.

Virtual, override, and abstract methods

You use virtual, override, and abstract methods to define the behavior for a hierarchy of class types. Because a class can derive from a base class, those derived class may need to modify the behavior implemented in the base class. A **virtual** method is one declared and implemented in a base class where any derived class may provide a more specific implementation. An **override** method is a method implemented in a derived class that modifies the behavior of the base class' implementation. An **abstract** method is a method declared in a base class that *must* be overridden in all derived classes. In fact, abstract methods don't define an implementation in the base class.

Method calls to instance methods may resolve to either base class or derived class implementations. The type of a variable determines its *compile-time type*. The *compile-time type* is the type the compiler uses to determine its members. However, a variable may be assigned to an instance of any type derived from its *compile-time type*. The *run-time type* is the type of the actual instance a variable refers to.

When a virtual method is invoked, the *run-time type* of the instance for which that invocation takes place determines the actual method implementation to invoke. In a nonvirtual method invocation, the *compile-time type* of the instance is the determining factor.

A virtual method can be *overridden* in a derived class. When an instance method declaration includes an override modifier, the method overrides an inherited virtual method with the same signature. A virtual method declaration introduces a new method. An override method declaration specializes an existing inherited virtual method by providing a new implementation of that method.

An *abstract method* is a virtual method with no implementation. An abstract method is declared with the `abstract` modifier and is permitted only in an abstract class. An abstract method must be overridden in every non-abstract derived class.

The following example declares an abstract class, `Expression`, which represents an expression tree node, and three derived classes, `Constant`, `VariableReference`, and `Operation`, which implement expression tree nodes for constants, variable references, and arithmetic operations. (This example is similar to, but not related to the expression tree types).

```

public abstract class Expression
{
    public abstract double Evaluate(Dictionary<string, object> vars);
}

public class Constant : Expression
{
    double _value;

    public Constant(double value)
    {
        _value = value;
    }

    public override double Evaluate(Dictionary<string, object> vars)
    {
        return _value;
    }
}

public class VariableReference : Expression
{
    string _name;

    public VariableReference(string name)
    {
        _name = name;
    }

    public override double Evaluate(Dictionary<string, object> vars)
    {
        object value = vars[_name] ?? throw new Exception($"Unknown variable: {_name}");
        return Convert.ToDouble(value);
    }
}

public class Operation : Expression
{
    Expression _left;
    char _op;
    Expression _right;

    public Operation(Expression left, char op, Expression right)
    {
        _left = left;
        _op = op;
        _right = right;
    }

    public override double Evaluate(Dictionary<string, object> vars)
    {
        double x = _left.Evaluate(vars);
        double y = _right.Evaluate(vars);
        switch (_op)
        {
            case '+': return x + y;
            case '-': return x - y;
            case '*': return x * y;
            case '/': return x / y;

            default: throw new Exception("Unknown operator");
        }
    }
}

```

The previous four classes can be used to model arithmetic expressions. For example, using instances of these

classes, the expression `x + 3` can be represented as follows.

```
Expression e = new Operation(  
    new VariableReference("x"),  
    '+',  
    new Constant(3));
```

The `Evaluate` method of an `Expression` instance is invoked to evaluate the given expression and produce a `double` value. The method takes a `Dictionary` argument that contains variable names (as keys of the entries) and values (as values of the entries). Because `Evaluate` is an abstract method, non-abstract classes derived from `Expression` must override `Evaluate`.

A `Constant`'s implementation of `Evaluate` simply returns the stored constant. A `VariableReference`'s implementation looks up the variable name in the dictionary and returns the resulting value. An `Operation`'s implementation first evaluates the left and right operands (by recursively invoking their `Evaluate` methods) and then performs the given arithmetic operation.

The following program uses the `Expression` classes to evaluate the expression `x * (y + 2)` for different values of `x` and `y`.

```
Expression e = new Operation(  
    new VariableReference("x"),  
    '*',  
    new Operation(  
        new VariableReference("y"),  
        '+',  
        new Constant(2)  
    )  
);  
Dictionary<string, object> vars = new();  
vars["x"] = 3;  
vars["y"] = 5;  
Console.WriteLine(e.Evaluate(vars)); // "21"  
vars["x"] = 1.5;  
vars["y"] = 9;  
Console.WriteLine(e.Evaluate(vars)); // "16.5"
```

Method overloading

Method *overloading* permits multiple methods in the same class to have the same name as long as they have unique signatures. When compiling an invocation of an overloaded method, the compiler uses *overload resolution* to determine the specific method to invoke. Overload resolution finds the one method that best matches the arguments. If no single best match can be found, an error is reported. The following example shows overload resolution in effect. The comment for each invocation in the `UsageExample` method shows which method is invoked.

```

class OverloadingExample
{
    static void F() => Console.WriteLine("F()");
    static void F(object x) => Console.WriteLine("F(object)");
    static void F(int x) => Console.WriteLine("F(int)");
    static void F(double x) => Console.WriteLine("F(double)");
    static void F<T>(T x) => Console.WriteLine("F<T>(T)");
    static void F(double x, double y) => Console.WriteLine("F(double, double)");

    public static void UsageExample()
    {
        F();           // Invokes F()
        F(1);          // Invokes F(int)
        F(1.0);        // Invokes F(double)
        F("abc");      // Invokes F<string>(string)
        F((double)1);  // Invokes F(double)
        F((object)1);  // Invokes F(object)
        F<int>(1);     // Invokes F<int>(int)
        F(1, 1);       // Invokes F(double, double)
    }
}

```

As shown by the example, a particular method can always be selected by explicitly casting the arguments to the exact parameter types and type arguments.

Other function members

Members that contain executable code are collectively known as the *function members* of a class. The preceding section describes methods, which are the primary types of function members. This section describes the other kinds of function members supported by C#: constructors, properties, indexers, events, operators, and finalizers.

The following example shows a generic class called `MyList<T>`, which implements a growable list of objects. The class contains several examples of the most common kinds of function members.

```

public class MyList<T>
{
    const int DefaultCapacity = 4;

    T[] _items;
    int _count;

    public MyList(int capacity = DefaultCapacity)
    {
        _items = new T[capacity];
    }

    public int Count => _count;

    public int Capacity
    {
        get => _items.Length;
        set
        {
            if (value < _count) value = _count;
            if (value != _items.Length)
            {
                T[] newItems = new T[value];
                Array.Copy(_items, 0, newItems, 0, _count);
                _items = newItems;
            }
        }
    }

    public T this[int index]

```

```

public class MyList<T>
{
    get => _items[index];
    set
    {
        _items[index] = value;
        OnChanged();
    }
}

public void Add(T item)
{
    if (_count == Capacity) Capacity = _count * 2;
    _items[_count] = item;
    _count++;
    OnChanged();
}

protected virtual void OnChanged() =>
    Changed?.Invoke(this, EventArgs.Empty);

public override bool Equals(object other) =>
    Equals(this, other as MyList<T>);

static bool Equals(MyList<T> a, MyList<T> b)
{
    if (Object.ReferenceEquals(a, null)) return Object.ReferenceEquals(b, null);
    if (Object.ReferenceEquals(b, null) || a._count != b._count)
        return false;
    for (int i = 0; i < a._count; i++)
    {
        if (!Object.Equals(a._items[i], b._items[i]))
        {
            return false;
        }
    }
    return true;
}

public event EventHandler Changed;

public static bool operator ==(MyList<T> a, MyList<T> b) =>
    Equals(a, b);

public static bool operator !=(MyList<T> a, MyList<T> b) =>
    !Equals(a, b);
}

```

Constructors

C# supports both instance and static constructors. An *instance constructor* is a member that implements the actions required to initialize an instance of a class. A *static constructor* is a member that implements the actions required to initialize a class itself when it's first loaded.

A constructor is declared like a method with no return type and the same name as the containing class. If a constructor declaration includes a `static` modifier, it declares a static constructor. Otherwise, it declares an instance constructor.

Instance constructors can be overloaded and can have optional parameters. For example, the `MyList<T>` class declares one instance constructor with a single optional `int` parameter. Instance constructors are invoked using the `new` operator. The following statements allocate two `MyList<string>` instances using the constructor of the `MyList` class with and without the optional argument.

```

MyList<string> list1 = new();
MyList<string> list2 = new(10);

```

Unlike other members, instance constructors aren't inherited. A class has no instance constructors other than those constructors actually declared in the class. If no instance constructor is supplied for a class, then an empty one with no parameters is automatically provided.

Properties

Properties are a natural extension of fields. Both are named members with associated types, and the syntax for accessing fields and properties is the same. However, unlike fields, properties don't denote storage locations. Instead, properties have *accessors* that specify the statements executed when their values are read or written. A *get accessor* reads the value. A *set accessor* writes the value.

A property is declared like a field, except that the declaration ends with a get accessor or a set accessor written between the delimiters `{` and `}` instead of ending in a semicolon. A property that has both a get accessor and a set accessor is a *read-write property*. A property that has only a get accessor is a *read-only property*. A property that has only a set accessor is a *write-only property*.

A get accessor corresponds to a parameterless method with a return value of the property type. A set accessor corresponds to a method with a single parameter named `value` and no return type. The get accessor computes the value of the property. The set accessor provides a new value for the property. When the property is the target of an assignment, or the operand of `++` or `--`, the set accessor is invoked. In other cases where the property is referenced, the get accessor is invoked.

The `MyList<T>` class declares two properties, `Count` and `Capacity`, which are read-only and read-write, respectively. The following code is an example of use of these properties:

```
MyList<string> names = new();
names.Capacity = 100; // Invokes set accessor
int i = names.Count;  // Invokes get accessor
int j = names.Capacity; // Invokes get accessor
```

Similar to fields and methods, C# supports both instance properties and static properties. Static properties are declared with the static modifier, and instance properties are declared without it.

The accessor(s) of a property can be virtual. When a property declaration includes a `virtual`, `abstract`, or `override` modifier, it applies to the accessor(s) of the property.

Indexers

An *indexer* is a member that enables objects to be indexed in the same way as an array. An indexer is declared like a property except that the name of the member is `this` followed by a parameter list written between the delimiters `[` and `]`. The parameters are available in the accessor(s) of the indexer. Similar to properties, indexers can be read-write, read-only, and write-only, and the accessor(s) of an indexer can be virtual.

The `MyList<T>` class declares a single read-write indexer that takes an `int` parameter. The indexer makes it possible to index `MyList<T>` instances with `int` values. For example:

```
MyList<string> names = new();
names.Add("Liz");
names.Add("Martha");
names.Add("Beth");
for (int i = 0; i < names.Count; i++)
{
    string s = names[i];
    names[i] = s.ToUpper();
}
```

Indexers can be overloaded. A class can declare multiple indexers as long as the number or types of their parameters differ.

Events

An *event* is a member that enables a class or object to provide notifications. An event is declared like a field except that the declaration includes an `event` keyword and the type must be a delegate type.

Within a class that declares an event member, the event behaves just like a field of a delegate type (provided the event isn't abstract and doesn't declare accessors). The field stores a reference to a delegate that represents the event handlers that have been added to the event. If no event handlers are present, the field is `null`.

The `MyList<T>` class declares a single event member called `Changed`, which indicates that a new item has been added to the list. The `Changed` event is raised by the `OnChanged` virtual method, which first checks whether the event is `null` (meaning that no handlers are present). The notion of raising an event is precisely equivalent to invoking the delegate represented by the event. There are no special language constructs for raising events.

Clients react to events through *event handlers*. Event handlers are attached using the `+=` operator and removed using the `-=` operator. The following example attaches an event handler to the `Changed` event of a `MyList<string>`.

```
class EventExample
{
    static int s_changeCount;

    static void ListChanged(object sender, EventArgs e)
    {
        s_changeCount++;
    }

    public static void Usage()
    {
        var names = new MyList<string>();
        names.Changed += new EventHandler(ListChanged);
        names.Add("Liz");
        names.Add("Martha");
        names.Add("Beth");
        Console.WriteLine(s_changeCount); // "3"
    }
}
```

For advanced scenarios where control of the underlying storage of an event is desired, an event declaration can explicitly provide `add` and `remove` accessors, which are similar to the `set` accessor of a property.

Operators

An *operator* is a member that defines the meaning of applying a particular expression operator to instances of a class. Three kinds of operators can be defined: unary operators, binary operators, and conversion operators. All operators must be declared as `public` and `static`.

The `MyList<T>` class declares two operators, `operator ==` and `operator !=`. These overridden operators give new meaning to expressions that apply those operators to `MyList` instances. Specifically, the operators define equality of two `MyList<T>` instances as comparing each of the contained objects using their `Equals` methods. The following example uses the `==` operator to compare two `MyList<int>` instances.

```

MyList<int> a = new();
a.Add(1);
a.Add(2);
MyList<int> b = new();
b.Add(1);
b.Add(2);
Console.WriteLine(a == b); // Outputs "True"
b.Add(3);
Console.WriteLine(a == b); // Outputs "False"

```

The first `Console.WriteLine` outputs `True` because the two lists contain the same number of objects with the same values in the same order. Had `MyList<T>` not defined `operator ==`, the first `Console.WriteLine` would have output `False` because `a` and `b` reference different `MyList<int>` instances.

Finalizers

A *finalizer* is a member that implements the actions required to finalize an instance of a class. Typically, a finalizer is needed to release unmanaged resources. Finalizers can't have parameters, they can't have accessibility modifiers, and they can't be invoked explicitly. The finalizer for an instance is invoked automatically during garbage collection. For more information, see the article on [finalizers](#).

The garbage collector is allowed wide latitude in deciding when to collect objects and run finalizers. Specifically, the timing of finalizer invocations isn't deterministic, and finalizers may be executed on any thread. For these and other reasons, classes should implement finalizers only when no other solutions are feasible.

The `using` statement provides a better approach to object destruction.

Expressions

Expressions are constructed from *operands* and *operators*. The operators of an expression indicate which operations to apply to the operands. Examples of operators include `+`, `-`, `*`, `/`, and `new`. Examples of operands include literals, fields, local variables, and expressions.

When an expression contains multiple operators, the *precedence* of the operators controls the order in which the individual operators are evaluated. For example, the expression `x + y * z` is evaluated as `x + (y * z)` because the `*` operator has higher precedence than the `+` operator.

When an operand occurs between two operators with the same precedence, the *associativity* of the operators controls the order in which the operations are performed:

- Except for the assignment and null-coalescing operators, all binary operators are *left-associative*, meaning that operations are performed from left to right. For example, `x + y + z` is evaluated as `(x + y) + z`.
- The assignment operators, the null-coalescing `??` and `??=` operators, and the conditional operator `?:` are *right-associative*, meaning that operations are performed from right to left. For example, `x = y = z` is evaluated as `x = (y = z)`.

Precedence and associativity can be controlled using parentheses. For example, `x + y * z` first multiplies `y` by `z` and then adds the result to `x`, but `(x + y) * z` first adds `x` and `y` and then multiplies the result by `z`.

Most operators can be *overloaded*. Operator overloading permits user-defined operator implementations to be specified for operations where one or both of the operands are of a user-defined class or struct type.

C# provides operators to perform [arithmetic](#), [logical](#), [bitwise and shift](#) operations and [equality](#) and [order](#) comparisons.

For the complete list of C# operators ordered by precedence level, see [C# operators](#).

Statements

The actions of a program are expressed using *statements*. C# supports several different kinds of statements, a number of which are defined in terms of embedded statements.

- A *block* permits multiple statements to be written in contexts where a single statement is allowed. A block consists of a list of statements written between the delimiters `{` and `}`.
- *Declaration statements* are used to declare local variables and constants.
- *Expression statements* are used to evaluate expressions. Expressions that can be used as statements include method invocations, object allocations using the `new` operator, assignments using `=` and the compound assignment operators, increment and decrement operations using the `++` and `--` operators and `await` expressions.
- *Selection statements* are used to select one of a number of possible statements for execution based on the value of some expression. This group contains the `if` and `switch` statements.
- *Iteration statements* are used to execute repeatedly an embedded statement. This group contains the `while`, `do`, `for`, and `foreach` statements.
- *Jump statements* are used to transfer control. This group contains the `break`, `continue`, `goto`, `throw`, `return`, and `yield` statements.
- The `try ... catch` statement is used to catch exceptions that occur during execution of a block, and the `try ... finally` statement is used to specify finalization code that is always executed, whether an exception occurred or not.
- The `checked` and `unchecked` statements are used to control the overflow-checking context for integral-type arithmetic operations and conversions.
- The `lock` statement is used to obtain the mutual-exclusion lock for a given object, execute a statement, and then release the lock.
- The `using` statement is used to obtain a resource, execute a statement, and then dispose of that resource.

The following lists the kinds of statements that can be used:

- Local variable declaration.
- Local constant declaration.
- Expression statement.
- `if` statement.
- `switch` statement.
- `while` statement.
- `do` statement.
- `for` statement.
- `foreach` statement.
- `break` statement.
- `continue` statement.
- `goto` statement.
- `return` statement.
- `yield` statement.
- `throw` statements and `try` statements.
- `checked` and `unchecked` statements.
- `lock` statement.
- `using` statement.

[PREVIOUS](#)[NEXT](#)

Major language areas

12/28/2021 • 9 minutes to read • [Edit Online](#)

Arrays, collections, and LINQ

C# and .NET provide many different collection types. Arrays have syntax defined by the language. Generic collection types are listed in the [System.Collections.Generic](#) namespace. Specialized collections include [System.Span<T>](#) for accessing continuous memory on the stack frame, and [System.Memory<T>](#) for accessing continuous memory on the managed heap. All collections, including arrays, [Span<T>](#), and [Memory<T>](#) share a unifying principle for iteration. You use the [System.Collections.Generic.IEnumerable<T>](#) interface. This unifying principle means that any of the collection types can be used with LINQ queries or other algorithms. You write methods using [IEnumerable<T>](#) and those algorithms work with any collection.

Arrays

An **array** is a data structure that contains a number of variables that are accessed through computed indices. The variables contained in an array, also called the *elements* of the array, are all of the same type. This type is called the *element type* of the array.

Array types are reference types, and the declaration of an array variable simply sets aside space for a reference to an array instance. Actual array instances are created dynamically at run time using the `new` operator. The `new` operation specifies the *length* of the new array instance, which is then fixed for the lifetime of the instance. The indices of the elements of an array range from `0` to `Length - 1`. The `new` operator automatically initializes the elements of an array to their default value, which, for example, is zero for all numeric types and `null` for all reference types.

The following example creates an array of `int` elements, initializes the array, and prints the contents of the array.

```
int[] a = new int[10];
for (int i = 0; i < a.Length; i++)
{
    a[i] = i * i;
}
for (int i = 0; i < a.Length; i++)
{
    Console.WriteLine($"a[{i}] = {a[i]}");
}
```

This example creates and operates on a *single-dimensional array*. C# also supports *multi-dimensional arrays*. The number of dimensions of an array type, also known as the *rank* of the array type, is one plus the number of commas between the square brackets of the array type. The following example allocates a single-dimensional, a two-dimensional, and a three-dimensional array, respectively.

```
int[] a1 = new int[10];
int[,] a2 = new int[10, 5];
int[,,] a3 = new int[10, 5, 2];
```

The `a1` array contains 10 elements, the `a2` array contains 50 (10×5) elements, and the `a3` array contains 100 ($10 \times 5 \times 2$) elements. The element type of an array can be any type, including an array type. An array with elements of an array type is sometimes called a *jagged array* because the lengths of the element arrays don't all have to be the same. The following example allocates an array of arrays of `int`:

```
int[][] a = new int[3][];
a[0] = new int[10];
a[1] = new int[5];
a[2] = new int[20];
```

The first line creates an array with three elements, each of type `int[]` and each with an initial value of `null`. The next lines then initialize the three elements with references to individual array instances of varying lengths.

The `new` operator permits the initial values of the array elements to be specified using an *array initializer*, which is a list of expressions written between the delimiters `{` and `}`. The following example allocates and initializes an `int[]` with three elements.

```
int[] a = new int[] { 1, 2, 3 };
```

The length of the array is inferred from the number of expressions between `{` and `}`. Array initialization can be shortened further such that the array type doesn't have to be restated.

```
int[] a = { 1, 2, 3 };
```

Both of the previous examples are equivalent to the following code:

```
int[] t = new int[3];
t[0] = 1;
t[1] = 2;
t[2] = 3;
int[] a = t;
```

The `foreach` statement can be used to enumerate the elements of any collection. The following code enumerates the array from the preceding example:

```
foreach (int item in a)
{
    Console.WriteLine(item);
}
```

The `foreach` statement uses the `IEnumerable<T>` interface, so can work with any collection.

String interpolation

C# *string interpolation* enables you to format strings by defining expressions whose results are placed in a format string. For example, the following example prints the temperature on a given day from a set of weather data:

```
Console.WriteLine($"The low and high temperature on {weatherData.Date:MM-DD-YYYY}");
Console.WriteLine($"    was {weatherData.LowTemp} and {weatherData.HighTemp}.");
// Output (similar to):
// The low and high temperature on 08-11-2020
//    was 5 and 30.
```

An interpolated string is declared using the `$` token. String interpolation evaluates the expressions between `{` and `}`, then converts the result to a `string`, and replaces the text between the brackets with the string result of the expression. The `:` in the first expression, `{weatherData.Date:MM-DD-YYYY}` specifies the *format string*. In the preceding example, it specifies that the date should be printed in "MM-DD-YYYY" format.

Pattern matching

The C# language provides *pattern matching* expressions to query the state of an object and execute code based on that state. You can inspect types and the values of properties and fields to determine which action to take. The `switch` expression is the primary expression for pattern matching.

Delegates and lambda expressions

A *delegate type* represents references to methods with a particular parameter list and return type. Delegates make it possible to treat methods as entities that can be assigned to variables and passed as parameters. Delegates are similar to the concept of function pointers found in some other languages. Unlike function pointers, delegates are object-oriented and type-safe.

The following example declares and uses a delegate type named `Function`.

```
delegate double Function(double x);

class Multiplier
{
    double _factor;

    public Multiplier(double factor) => _factor = factor;

    public double Multiply(double x) => x * _factor;
}

class DelegateExample
{
    static double[] Apply(double[] a, Function f)
    {
        var result = new double[a.Length];
        for (int i = 0; i < a.Length; i++) result[i] = f(a[i]);
        return result;
    }

    public static void Main()
    {
        double[] a = { 0.0, 0.5, 1.0 };
        double[] squares = Apply(a, (x) => x * x);
        double[] sines = Apply(a, Math.Sin);
        Multiplier m = new(2.0);
        double[] doubles = Apply(a, m.Multiply);
    }
}
```

An instance of the `Function` delegate type can reference any method that takes a `double` argument and returns a `double` value. The `Apply` method applies a given `Function` to the elements of a `double[]`, returning a `double[]` with the results. In the `Main` method, `Apply` is used to apply three different functions to a `double[]`.

A delegate can reference either a static method (such as `Square` or `Math.Sin` in the previous example) or an instance method (such as `m.Multiply` in the previous example). A delegate that references an instance method also references a particular object, and when the instance method is invoked through the delegate, that object becomes `this` in the invocation.

Delegates can also be created using anonymous functions or lambda expressions, which are "inline methods" that are created when declared. Anonymous functions can see the local variables of the surrounding methods. The following example doesn't create a class:

```
double[] doubles = Apply(a, (double x) => x * 2.0);
```

A delegate doesn't know or care about the class of the method it references. The referenced method must have the same parameters and return type as the delegate.

async / await

C# supports asynchronous programs with two keywords: `async` and `await`. You add the `async` modifier to a method declaration to declare the method is asynchronous. The `await` operator tells the compiler to asynchronously await for a result to finish. Control is returned to the caller, and the method returns a structure that manages the state of the asynchronous work. The structure is typically a [System.Threading.Tasks.Task<TResult>](#), but can be any type that supports the awaiter pattern. These features enable you to write code that reads as its synchronous counterpart, but executes asynchronously. For example, the following code downloads the home page for [Microsoft docs](#):

```
public async Task<int> RetrieveDocsHomePage()
{
    var client = new HttpClient();
    byte[] content = await client.GetByteArrayAsync("https://docs.microsoft.com/");

    Console.WriteLine($"{nameof(RetrieveDocsHomePage)}: Finished downloading.");
    return content.Length;
}
```

This small sample shows the major features for asynchronous programming:

- The method declaration includes the `async` modifier.
- The body of the method `await`s the return of the `GetByteArrayAsync` method.
- The type specified in the `return` statement matches the type argument in the `Task<T>` declaration for the method. (A method that returns a `Task` would use `return` statements without any argument).

Attributes

Types, members, and other entities in a C# program support modifiers that control certain aspects of their behavior. For example, the accessibility of a method is controlled using the `public`, `protected`, `internal`, and `private` modifiers. C# generalizes this capability such that user-defined types of declarative information can be attached to program entities and retrieved at run-time. Programs specify this declarative information by defining and using [attributes](#).

The following example declares a `HelpAttribute` attribute that can be placed on program entities to provide links to their associated documentation.

```
public class HelpAttribute : Attribute
{
    string _url;
    string _topic;

    public HelpAttribute(string url) => _url = url;

    public string Url => _url;

    public string Topic
    {
        get => _topic;
        set => _topic = value;
    }
}
```

All attribute classes derive from the [Attribute](#) base class provided by the .NET library. Attributes can be applied

by giving their name, along with any arguments, inside square brackets just before the associated declaration. If an attribute's name ends in `Attribute`, that part of the name can be omitted when the attribute is referenced. For example, the `HelpAttribute` can be used as follows.

```
[Help("https://docs.microsoft.com/dotnet/csharp/tour-of-csharp/features")]
public class Widget
{
    [Help("https://docs.microsoft.com/dotnet/csharp/tour-of-csharp/features",
        Topic = "Display")]
    public void Display(string text) { }
}
```

This example attaches a `HelpAttribute` to the `Widget` class. It adds another `HelpAttribute` to the `Display` method in the class. The public constructors of an attribute class control the information that must be provided when the attribute is attached to a program entity. Additional information can be provided by referencing public read-write properties of the attribute class (such as the reference to the `Topic` property previously).

The metadata defined by attributes can be read and manipulated at run time using reflection. When a particular attribute is requested using this technique, the constructor for the attribute class is invoked with the information provided in the program source. The resulting attribute instance is returned. If additional information was provided through properties, those properties are set to the given values before the attribute instance is returned.

The following code sample demonstrates how to get the `HelpAttribute` instances associated to the `Widget` class and its `Display` method.

```
Type widgetType = typeof(Widget);

object[] widgetClassAttributes = widgetType.GetCustomAttributes(typeof(HelpAttribute), false);

if (widgetClassAttributes.Length > 0)
{
    HelpAttribute attr = (HelpAttribute)widgetClassAttributes[0];
    Console.WriteLine($"Widget class help URL : {attr.Url} - Related topic : {attr.Topic}");
}

System.Reflection.MethodInfo displayMethod = widgetType.GetMethod(nameof(Widget.Display));

object[] displayMethodAttributes = displayMethod.GetCustomAttributes(typeof(HelpAttribute), false);

if (displayMethodAttributes.Length > 0)
{
    HelpAttribute attr = (HelpAttribute)displayMethodAttributes[0];
    Console.WriteLine($"Display method help URL : {attr.Url} - Related topic : {attr.Topic}");
}
```

Learn more

You can explore more about C# by trying one of our [tutorials](#).

PREVIOUS

Introduction to C#

12/28/2021 • 2 minutes to read • [Edit Online](#)

Welcome to the introduction to C# tutorials. These lessons start with interactive code that you can run in your browser. You can learn the basics of C# from the [C# 101 video series](#) before starting these interactive lessons.

The first lessons explain C# concepts using small snippets of code. You'll learn the basics of C# syntax and how to work with data types like strings, numbers, and booleans. It's all interactive, and you'll be writing and running code within minutes. These first lessons assume no prior knowledge of programming or the C# language.

You can try these tutorials in different environments. The concepts you'll learn are the same. The difference is which experience you prefer:

- [In your browser, on the docs platform](#): This experience embeds a runnable C# code window in docs pages. You write and execute C# code in the browser.
- [In the Microsoft Learn experience](#). This learning path contains several modules that teach the basics of C#.
- [In Jupyter on Binder](#). You can experiment with C# code in a Jupyter notebook on binder.
- [On your local machine](#). After you've explored online, you can [download](#) the .NET SDK and build programs on your machine.

All the introductory tutorials following the Hello World lesson are available using the online browser experience or [in your own local development environment](#). At the end of each tutorial, you decide if you want to continue with the next lesson online or on your own machine. There are links to help you set up your environment and continue with the next tutorial on your machine.

Hello world

In the [Hello world](#) tutorial, you'll create the most basic C# program. You'll explore the `string` type and how to work with text. You can also use the path on [Microsoft Learn](#) or [Jupyter on Binder](#).

Numbers in C#

In the [Numbers in C#](#) tutorial, you'll learn how computers store numbers and how to perform calculations with different numeric types. You'll learn the basics of rounding, and how to perform mathematical calculations using C#. This tutorial is also available [to run locally on your machine](#).

This tutorial assumes that you've finished the [Hello world](#) lesson.

Branches and loops

The [Branches and loops](#) tutorial teaches the basics of selecting different paths of code execution based on the values stored in variables. You'll learn the basics of control flow, which is the basis of how programs make decisions and choose different actions. This tutorial is also available [to run locally on your machine](#).

This tutorial assumes that you've finished the [Hello world](#) and [Numbers in C#](#) lessons.

List collection

The [List collection](#) lesson gives you a tour of the List collection type that stores sequences of data. You'll learn how to add and remove items, search for items, and sort the lists. You'll explore different kinds of lists. This

tutorial is also available [to run locally on your machine](#).

This tutorial assumes that you've finished the lessons listed above.

101 Linq Samples

This sample requires the [dotnet-try](#) global tool. Once you install the tool, and clone the [try-samples](#) repo, you can learn Language Integrated Query (LINQ) through a set of 101 samples you can run interactively. You can explore different ways to query, explore, and transform data sequences.

Set up your local environment

12/28/2021 • 2 minutes to read • [Edit Online](#)

The first step in running a tutorial on your machine is to set up a development environment. Choose one of the following alternatives:

- To use the .NET CLI and your choice of text or code editor, see the .NET tutorial [Hello World in 10 minutes](#). The tutorial has instructions for setting up a development environment on Windows, Linux, or macOS.
- To use the .NET CLI and Visual Studio Code, install the [.NET SDK](#) and [Visual Studio Code](#).
- To use Visual Studio 2019, see [Tutorial: Create a simple C# console app in Visual Studio](#).

Basic application development flow

The instructions in these tutorials assume that you're using the .NET CLI to create, build, and run applications. You'll use the following commands:

- `dotnet new` creates an application. This command generates the files and assets necessary for your application. The introduction to C# tutorials all use the `console` application type. Once you've got the basics, you can expand to other application types.
- `dotnet build` builds the executable.
- `dotnet run` runs the executable.

If you use Visual Studio 2019 for these tutorials, you'll choose a Visual Studio menu selection when a tutorial directs you to run one of these CLI commands:

- **File > New > Project** creates an application.
 - The `Console Application` project template is recommended.
 - You will be given the option to specify a target framework. The tutorials below work best when targeting .NET 5 or higher.
- **Build > Build Solution** builds the executable.
- **Debug > Start Without Debugging** runs the executable.

Pick your tutorial

You can start with any of the following tutorials:

Numbers in C#

In the [Numbers in C#](#) tutorial, you'll learn how computers store numbers and how to perform calculations with different numeric types. You'll learn the basics of rounding and how to perform mathematical calculations using C#.

This tutorial assumes that you have finished the [Hello world](#) lesson.

Branches and loops

The [Branches and loops](#) tutorial teaches the basics of selecting different paths of code execution based on the values stored in variables. You'll learn the basics of control flow, which is the basis of how programs make decisions and choose different actions.

This tutorial assumes that you have finished the [Hello world](#) and [Numbers in C#](#) lessons.

List collection

The [List collection](#) lesson gives you a tour of the List collection type that stores sequences of data. You'll learn how to add and remove items, search for items, and sort the lists. You'll explore different kinds of lists.

This tutorial assumes that you have finished the lessons listed above.

Manipulate integral and floating point numbers in C#

12/28/2021 • 9 minutes to read • [Edit Online](#)

This tutorial teaches you about the numeric types in C# interactively. You'll write small amounts of code, then you'll compile and run that code. The tutorial contains a series of lessons that explore numbers and math operations in C#. These lessons teach you the fundamentals of the C# language.

Prerequisites

The tutorial expects that you have a machine set up for local development. On Windows, Linux, or macOS, you can use the .NET CLI to create, build, and run applications. On Mac or Windows, you can use Visual Studio 2019. For setup instructions, see [Set up your local environment](#).

Explore integer math

Create a directory named *numbers-quickstart*. Make it the current directory and run the following command:

```
dotnet new console -n NumbersInCSharp -o .
```

IMPORTANT

The C# templates for .NET 6 use *top level statements*. Your application may not match the code in this article, if you've already upgraded to the .NET 6 previews. For more information see the article on [New C# templates generate top level statements](#)

The .NET 6 SDK also adds a set of *implicit* `global using` directives for projects that use the following SDKs:

- Microsoft.NET.Sdk
- Microsoft.NET.Sdk.Web
- Microsoft.NET.Sdk.Worker

These implicit `global using` directives include the most common namespaces for the project type.

Open *Program.cs* in your favorite editor, and replace the contents of the file with the following code:

```
using System;

int a = 18;
int b = 6;
int c = a + b;
Console.WriteLine(c);
```

Run this code by typing `dotnet run` in your command window.

You've seen one of the fundamental math operations with integers. The `int` type represents an **integer**, a zero, positive, or negative whole number. You use the `+` symbol for addition. Other common mathematical operations for integers include:

- `-` for subtraction

- `*` for multiplication
- `/` for division

Start by exploring those different operations. Add these lines after the line that writes the value of `c`:

```
// subtraction
c = a - b;
Console.WriteLine(c);

// multiplication
c = a * b;
Console.WriteLine(c);

// division
c = a / b;
Console.WriteLine(c);
```

Run this code by typing `dotnet run` in your command window.

You can also experiment by writing multiple mathematics operations in the same line, if you'd like. Try

`c = a + b - 12 * 17;` for example. Mixing variables and constant numbers is allowed.

TIP

As you explore C# (or any programming language), you'll make mistakes when you write code. The **compiler** will find those errors and report them to you. When the contains error messages, look closely at the example code and the code in your window to see what to fix. That exercise will help you learn the structure of C# code.

You've finished the first step. Before you start the next section, let's move the current code into a separate *method*. A method is a series of statements grouped together and given a name. You call a method by writing the method's name followed by `()`. Organizing your code into methods makes it easier to start working with a new example. When you finish, your code should look like this:

```
using System;

WorkWithIntegers();

void WorkWithIntegers()
{
    int a = 18;
    int b = 6;
    int c = a + b;
    Console.WriteLine(c);

    // subtraction
    c = a - b;
    Console.WriteLine(c);

    // multiplication
    c = a * b;
    Console.WriteLine(c);

    // division
    c = a / b;
    Console.WriteLine(c);
}
```

The line `WorkWithIntegers();` invokes the method. The following code declares the method and defines it.

Explore order of operations

Comment out the call to `WorkingWithIntegers()`. It will make the output less cluttered as you work in this section:

```
//WorkingWithIntegers();
```

The `//` starts a **comment** in C#. Comments are any text you want to keep in your source code but not execute as code. The compiler doesn't generate any executable code from comments. Because `WorkingWithIntegers()` is a method, you need to only comment out one line.

The C# language defines the precedence of different mathematics operations with rules consistent with the rules you learned in mathematics. Multiplication and division take precedence over addition and subtraction. Explore that by adding the following code after the call to `WorkingWithIntegers()`, and executing `dotnet run`:

```
int a = 5;
int b = 4;
int c = 2;
int d = a + b * c;
Console.WriteLine(d);
```

The output demonstrates that the multiplication is performed before the addition.

You can force a different order of operation by adding parentheses around the operation or operations you want performed first. Add the following lines and run again:

```
d = (a + b) * c;
Console.WriteLine(d);
```

Explore more by combining many different operations. Add something like the following lines. Try `dotnet run` again.

```
d = (a + b) - 6 * c + (12 * 4) / 3 + 12;
Console.WriteLine(d);
```

You may have noticed an interesting behavior for integers. Integer division always produces an integer result, even when you'd expect the result to include a decimal or fractional portion.

If you haven't seen this behavior, try the following code:

```
int e = 7;
int f = 4;
int g = 3;
int h = (e + f) / g;
Console.WriteLine(h);
```

Type `dotnet run` again to see the results.

Before moving on, let's take all the code you've written in this section and put it in a new method. Call that new method `OrderPrecedence`. Your code should look something like this:


```

using System;

// WorkWithIntegers();
OrderPrecedence();

void WorkWithIntegers()
{
    int a = 18;
    int b = 6;
    int c = a + b;
    Console.WriteLine(c);

    // subtraction
    c = a - b;
    Console.WriteLine(c);

    // multiplication
    c = a * b;
    Console.WriteLine(c);

    // division
    c = a / b;
    Console.WriteLine(c);
}

void OrderPrecedence()
{
    int a = 5;
    int b = 4;
    int c = 2;
    int d = a + b * c;
    Console.WriteLine(d);

    d = (a + b) * c;
    Console.WriteLine(d);

    d = (a + b) - 6 * c + (12 * 4) / 3 + 12;
    Console.WriteLine(d);

    int e = 7;
    int f = 4;
    int g = 3;
    int h = (e + f) / g;
    Console.WriteLine(h);
}

```

Explore integer precision and limits

That last sample showed you that integer division truncates the result. You can get the **remainder** by using the **modulo** operator, the `%` character. Try the following code after the method call to `OrderPrecedence()`:

```

int a = 7;
int b = 4;
int c = 3;
int d = (a + b) / c;
int e = (a + b) % c;
Console.WriteLine($"quotient: {d}");
Console.WriteLine($"remainder: {e}");

```

The C# integer type differs from mathematical integers in one other way: the `int` type has minimum and maximum limits. Add this code to see those limits:

```
int max = int.MaxValue;
int min = int.MinValue;
Console.WriteLine($"The range of integers is {min} to {max}");
```

If a calculation produces a value that exceeds those limits, you have an **underflow** or **overflow** condition. The answer appears to wrap from one limit to the other. Add these two lines to see an example:

```
int what = max + 3;
Console.WriteLine($"An example of overflow: {what}");
```

Notice that the answer is very close to the minimum (negative) integer. It's the same as `min + 2`. The addition operation **overflowed** the allowed values for integers. The answer is a very large negative number because an overflow "wraps around" from the largest possible integer value to the smallest.

There are other numeric types with different limits and precision that you would use when the `int` type doesn't meet your needs. Let's explore those other types next. Before you start the next section, move the code you wrote in this section into a separate method. Name it `TestLimits`.

Work with the double type

The `double` numeric type represents a double-precision floating point number. Those terms may be new to you. A **floating point** number is useful to represent non-integral numbers that may be very large or small in magnitude. **Double-precision** is a relative term that describes the number of binary digits used to store the value. **Double precision** numbers have twice the number of binary digits as **single-precision**. On modern computers, it's more common to use double precision than single precision numbers. **Single precision** numbers are declared using the `float` keyword. Let's explore. Add the following code and see the result:

```
double a = 5;
double b = 4;
double c = 2;
double d = (a + b) / c;
Console.WriteLine(d);
```

Notice that the answer includes the decimal portion of the quotient. Try a slightly more complicated expression with doubles:

```
double e = 19;
double f = 23;
double g = 8;
double h = (e + f) / g;
Console.WriteLine(h);
```

The range of a double value is much greater than integer values. Try the following code below what you've written so far:

```
double max = double.MaxValue;
double min = double.MinValue;
Console.WriteLine($"The range of double is {min} to {max}");
```

These values are printed in scientific notation. The number to the left of the `E` is the significand. The number to the right is the exponent, as a power of 10. Just like decimal numbers in math, doubles in C# can have rounding errors. Try this code:

```
double third = 1.0 / 3.0;
Console.WriteLine(third);
```

You know that `0.3` repeating isn't exactly the same as `1/3`.

Challenge

Try other calculations with large numbers, small numbers, multiplication, and division using the `double` type. Try more complicated calculations. After you've spent some time with the challenge, take the code you've written and place it in a new method. Name that new method `WorkWithDoubles`.

Work with decimal types

You've seen the basic numeric types in C#: integers and doubles. There's one other type to learn: the `decimal` type. The `decimal` type has a smaller range but greater precision than `double`. Let's take a look:

```
decimal min = decimal.MinValue;
decimal max = decimal.MaxValue;
Console.WriteLine($"The range of the decimal type is {min} to {max}");
```

Notice that the range is smaller than the `double` type. You can see the greater precision with the decimal type by trying the following code:

```
double a = 1.0;
double b = 3.0;
Console.WriteLine(a / b);

decimal c = 1.0M;
decimal d = 3.0M;
Console.WriteLine(c / d);
```

The `M` suffix on the numbers is how you indicate that a constant should use the `decimal` type. Otherwise, the compiler assumes the `double` type.

NOTE

The letter `M` was chosen as the most visually distinct letter between the `double` and `decimal` keywords.

Notice that the math using the decimal type has more digits to the right of the decimal point.

Challenge

Now that you've seen the different numeric types, write code that calculates the area of a circle whose radius is 2.50 centimeters. Remember that the area of a circle is the radius squared multiplied by PI. One hint: .NET contains a constant for PI, `Math.PI` that you can use for that value. `Math.PI`, like all constants declared in the `System.Math` namespace, is a `double` value. For that reason, you should use `double` instead of `decimal` values for this challenge.

You should get an answer between 19 and 20. You can check your answer by [looking at the finished sample code on GitHub](#).

Try some other formulas if you'd like.

You've completed the "Numbers in C#" quickstart. You can continue with the [Branches and loops](#) quickstart in your own development environment.

You can learn more about numbers in C# in the following articles:

- [Integral numeric types](#)
- [Floating-point numeric types](#)
- [Built-in numeric conversions](#)

Learn conditional logic with branch and loop statements

12/28/2021 • 10 minutes to read • [Edit Online](#)

This tutorial teaches you how to write code that examines variables and changes the execution path based on those variables. You write C# code and see the results of compiling and running it. The tutorial contains a series of lessons that explore branching and looping constructs in C#. These lessons teach you the fundamentals of the C# language.

Prerequisites

The tutorial expects that you have a machine set up for local development. On Windows, Linux, or macOS, you can use the .NET CLI to create, build, and run applications. On Mac and Windows, you can use Visual Studio 2019. For setup instructions, see [Set up your local environment](#).

Make decisions using the `if` statement

Create a directory named *branches-tutorial*. Make that the current directory and run the following command:

```
dotnet new console -n BranchesAndLoops -o .
```

IMPORTANT

The C# templates for .NET 6 use *top level statements*. Your application may not match the code in this article, if you've already upgraded to the .NET 6 previews. For more information see the article on [New C# templates generate top level statements](#)

The .NET 6 SDK also adds a set of *implicit* `global using` directives for projects that use the following SDKs:

- Microsoft.NET.Sdk
- Microsoft.NET.Sdk.Web
- Microsoft.NET.Sdk.Worker

These implicit `global using` directives include the most common namespaces for the project type.

This command creates a new .NET console application in the current directory. Open *Program.cs* in your favorite editor, and replace the contents with the following code:

```
using System;

int a = 5;
int b = 6;
if (a + b > 10)
    Console.WriteLine("The answer is greater than 10.");
```

Try this code by typing `dotnet run` in your console window. You should see the message "The answer is greater than 10." printed to your console. Modify the declaration of `b` so that the sum is less than 10:

```
int b = 3;
```

Type `dotnet run` again. Because the answer is less than 10, nothing is printed. The **condition** you're testing is false. You don't have any code to execute because you've only written one of the possible branches for an `if` statement: the true branch.

TIP

As you explore C# (or any programming language), you'll make mistakes when you write code. The compiler will find and report the errors. Look closely at the error output and the code that generated the error. The compiler error can usually help you find the problem.

This first sample shows the power of `if` and Boolean types. A *Boolean* is a variable that can have one of two values: `true` or `false`. C# defines a special type, `bool` for Boolean variables. The `if` statement checks the value of a `bool`. When the value is `true`, the statement following the `if` executes. Otherwise, it's skipped. This process of checking conditions and executing statements based on those conditions is powerful.

Make if and else work together

To execute different code in both the true and false branches, you create an `else` branch that executes when the condition is false. Try an `else` branch. Add the last two lines in the code below (you should already have the first four):

```
int a = 5;
int b = 3;
if (a + b > 10)
    Console.WriteLine("The answer is greater than 10");
else
    Console.WriteLine("The answer is not greater than 10");
```

The statement following the `else` keyword executes only when the condition being tested is `false`. Combining `if` and `else` with Boolean conditions provides all the power you need to handle both a `true` and a `false` condition.

IMPORTANT

The indentation under the `if` and `else` statements is for human readers. The C# language doesn't treat indentation or white space as significant. The statement following the `if` or `else` keyword will be executed based on the condition. All the samples in this tutorial follow a common practice to indent lines based on the control flow of statements.

Because indentation isn't significant, you need to use `{` and `}` to indicate when you want more than one statement to be part of the block that executes conditionally. C# programmers typically use those braces on all `if` and `else` clauses. The following example is the same as the one you created. Modify your code above to match the following code:

```
int a = 5;
int b = 3;
if (a + b > 10)
{
    Console.WriteLine("The answer is greater than 10");
}
else
{
    Console.WriteLine("The answer is not greater than 10");
}
```

TIP

Through the rest of this tutorial, the code samples all include the braces, following accepted practices.

You can test more complicated conditions. Add the following code after the code you've written so far:

```
int c = 4;
if ((a + b + c > 10) && (a == b))
{
    Console.WriteLine("The answer is greater than 10");
    Console.WriteLine("And the first number is equal to the second");
}
else
{
    Console.WriteLine("The answer is not greater than 10");
    Console.WriteLine("Or the first number is not equal to the second");
}
```

The `==` symbol tests for *equality*. Using `==` distinguishes the test for equality from assignment, which you saw in `a = 5`.

The `&&` represents "and". It means both conditions must be true to execute the statement in the true branch. These examples also show that you can have multiple statements in each conditional branch, provided you enclose them in `{` and `}`. You can also use `||` to represent "or". Add the following code after what you've written so far:

```
if ((a + b + c > 10) || (a == b))
{
    Console.WriteLine("The answer is greater than 10");
    Console.WriteLine("Or the first number is equal to the second");
}
else
{
    Console.WriteLine("The answer is not greater than 10");
    Console.WriteLine("And the first number is not equal to the second");
}
```

Modify the values of `a`, `b`, and `c` and switch between `&&` and `||` to explore. You'll gain more understanding of how the `&&` and `||` operators work.

You've finished the first step. Before you start the next section, let's move the current code into a separate method. That makes it easier to start working with a new example. Put the existing code in a method called `ExploreIf()`. Call it from the top of your program. When you finished those changes, your code should look like the following:

```

using System;

ExploreIf();

void ExploreIf()
{
    int a = 5;
    int b = 3;
    if (a + b > 10)
    {
        Console.WriteLine("The answer is greater than 10");
    }
    else
    {
        Console.WriteLine("The answer is not greater than 10");
    }

    int c = 4;
    if ((a + b + c > 10) && (a > b))
    {
        Console.WriteLine("The answer is greater than 10");
        Console.WriteLine("And the first number is greater than the second");
    }
    else
    {
        Console.WriteLine("The answer is not greater than 10");
        Console.WriteLine("Or the first number is not greater than the second");
    }

    if ((a + b + c > 10) || (a > b))
    {
        Console.WriteLine("The answer is greater than 10");
        Console.WriteLine("Or the first number is greater than the second");
    }
    else
    {
        Console.WriteLine("The answer is not greater than 10");
        Console.WriteLine("And the first number is not greater than the second");
    }
}

```

Comment out the call to `ExploreIf()`. It will make the output less cluttered as you work in this section:

```
//ExploreIf();
```

The `//` starts a **comment** in C#. Comments are any text you want to keep in your source code but not execute as code. The compiler doesn't generate any executable code from comments.

Use loops to repeat operations

In this section, you use **loops** to repeat statements. Add this code after the call to `ExploreIf` :

```

int counter = 0;
while (counter < 10)
{
    Console.WriteLine($"Hello World! The counter is {counter}");
    counter++;
}

```

The `while` statement checks a condition and executes the statement or statement block following the `while`. It repeatedly checks the condition and executing those statements until the condition is false.

There's one other new operator in this example. The `++` after the `counter` variable is the **increment** operator. It adds 1 to the value of `counter` and stores that value in the `counter` variable.

IMPORTANT

Make sure that the `while` loop condition changes to false as you execute the code. Otherwise, you create an **infinite loop** where your program never ends. That is not demonstrated in this sample, because you have to force your program to quit using **CTRL-C** or other means.

The `while` loop tests the condition before executing the code following the `while`. The `do ... while` loop executes the code first, and then checks the condition. The *do while* loop is shown in the following code:

```
int counter = 0;
do
{
    Console.WriteLine($"Hello World! The counter is {counter}");
    counter++;
} while (counter < 10);
```

This `do` loop and the earlier `while` loop produce the same output.

Work with the for loop

The `for` loop is commonly used in C#. Try this code:

```
for (int index = 0; index < 10; index++)
{
    Console.WriteLine($"Hello World! The index is {index}");
}
```

The previous code does the same work as the `while` loop and the `do` loop you've already used. The `for` statement has three parts that control how it works.

The first part is the **for initializer**: `int index = 0;` declares that `index` is the loop variable, and sets its initial value to `0`.

The middle part is the **for condition**: `index < 10` declares that this `for` loop continues to execute as long as the value of counter is less than 10.

The final part is the **for iterator**: `index++` specifies how to modify the loop variable after executing the block following the `for` statement. Here, it specifies that `index` should be incremented by 1 each time the block executes.

Experiment yourself. Try each of the following variations:

- Change the initializer to start at a different value.
- Change the condition to stop at a different value.

When you're done, let's move on to write some code yourself to use what you've learned.

There's one other looping statement that isn't covered in this tutorial: the `foreach` statement. The `foreach` statement repeats its statement for every item in a sequence of items. It's most often used with *collections*, so it's covered in the next tutorial.

Created nested loops

A `while`, `do`, or `for` loop can be nested inside another loop to create a matrix using the combination of each item in the outer loop with each item in the inner loop. Let's do that to build a set of alphanumeric pairs to represent rows and columns.

One `for` loop can generate the rows:

```
for (int row = 1; row < 11; row++)
{
    Console.WriteLine($"The row is {row}");
}
```

Another loop can generate the columns:

```
for (char column = 'a'; column < 'k'; column++)
{
    Console.WriteLine($"The column is {column}");
}
```

You can nest one loop inside the other to form pairs:

```
for (int row = 1; row < 11; row++)
{
    for (char column = 'a'; column < 'k'; column++)
    {
        Console.WriteLine($"The cell is ({row}, {column})");
    }
}
```

You can see that the outer loop increments once for each full run of the inner loop. Reverse the row and column nesting, and see the changes for yourself. When you're done, place the code from this section in a method called `ExploreLoops()`.

Combine branches and loops

Now that you've seen the `if` statement and the looping constructs in the C# language, see if you can write C# code to find the sum of all integers 1 through 20 that are divisible by 3. Here are a few hints:

- The `%` operator gives you the remainder of a division operation.
- The `if` statement gives you the condition to see if a number should be part of the sum.
- The `for` loop can help you repeat a series of steps for all the numbers 1 through 20.

Try it yourself. Then check how you did. You should get 63 for an answer. You can see one possible answer by [viewing the completed code on GitHub](#).

You've completed the "branches and loops" tutorial.

You can continue with the [Arrays and collections](#) tutorial in your own development environment.

You can learn more about these concepts in these articles:

- [Selection statements](#)
- [Iteration statements](#)

Learn to manage data collections using the generic list type

12/28/2021 • 6 minutes to read • [Edit Online](#)

This introductory tutorial provides an introduction to the C# language and the basics of the `List<T>` class.

Prerequisites

The tutorial expects that you have a machine set up for local development. On Windows, Linux, or macOS, you can use the .NET CLI to create, build, and run applications. On Mac and Windows, you can use Visual Studio 2019. For setup instructions, see [Set up your local environment](#).

A basic list example

Create a directory named *list-tutorial*. Make that the current directory and run `dotnet new console`.

IMPORTANT

The C# templates for .NET 6 use *top level statements*. Your application may not match the code in this article, if you've already upgraded to the .NET 6 previews. For more information see the article on [New C# templates generate top level statements](#)

The .NET 6 SDK also adds a set of *implicit* `global using` directives for projects that use the following SDKs:

- Microsoft.NET.Sdk
- Microsoft.NET.Sdk.Web
- Microsoft.NET.Sdk.Worker

These implicit `global using` directives include the most common namespaces for the project type.

Open *Program.cs* in your favorite editor, and replace the existing code with the following:

```
using System;
using System.Collections.Generic;

var names = new List<string> { "<name>", "Ana", "Felipe" };
foreach (var name in names)
{
    Console.WriteLine($"Hello {name.ToUpper()}!");
}
```

Replace `<name>` with your name. Save *Program.cs*. Type `dotnet run` in your console window to try it.

You've created a list of strings, added three names to that list, and printed the names in all CAPS. You're using concepts that you've learned in earlier tutorials to loop through the list.

The code to display names makes use of the [string interpolation](#) feature. When you precede a `string` with the `$` character, you can embed C# code in the string declaration. The actual string replaces that C# code with the value it generates. In this example, it replaces the `{name.ToUpper()}` with each name, converted to capital letters, because you called the [ToUpper](#) method.

Let's keep exploring.

Modify list contents

The collection you created uses the [List<T>](#) type. This type stores sequences of elements. You specify the type of the elements between the angle brackets.

One important aspect of this [List<T>](#) type is that it can grow or shrink, enabling you to add or remove elements. Add this code at the end of your program:

```
Console.WriteLine();
names.Add("Maria");
names.Add("Bill");
names.Remove("Ana");
foreach (var name in names)
{
    Console.WriteLine($"Hello {name.ToUpper()}!");
}
```

You've added two more names to the end of the list. You've also removed one as well. Save the file, and type `dotnet run` to try it.

The [List<T>](#) enables you to reference individual items by **index** as well. You place the index between `[` and `]` tokens following the list name. C# uses 0 for the first index. Add this code directly below the code you just added and try it:

```
Console.WriteLine($"My name is {names[0]}");
Console.WriteLine($"I've added {names[2]} and {names[3]} to the list");
```

You can't access an index beyond the end of the list. Remember that indices start at 0, so the largest valid index is one less than the number of items in the list. You can check how long the list is using the [Count](#) property. Add the following code at the end of your program:

```
Console.WriteLine($"The list has {names.Count} people in it");
```

Save the file, and type `dotnet run` again to see the results.

Search and sort lists

Our samples use relatively small lists, but your applications may often create lists with many more elements, sometimes numbering in the thousands. To find elements in these larger collections, you need to search the list for different items. The [IndexOf](#) method searches for an item and returns the index of the item. If the item isn't in the list, `IndexOf` returns `-1`. Add this code to the bottom of your program:

```

var index = names.IndexOf("Felipe");
if (index == -1)
{
    Console.WriteLine($"When an item is not found, IndexOf returns {index}");
}
else
{
    Console.WriteLine($"The name {names[index]} is at index {index}");
}

index = names.IndexOf("Not Found");
if (index == -1)
{
    Console.WriteLine($"When an item is not found, IndexOf returns {index}");
}
else
{
    Console.WriteLine($"The name {names[index]} is at index {index}");
}

```

The items in your list can be sorted as well. The [Sort](#) method sorts all the items in the list in their normal order (alphabetically for strings). Add this code to the bottom of your program:

```

names.Sort();
foreach (var name in names)
{
    Console.WriteLine($"Hello {name.ToUpper()}!");
}

```

Save the file and type `dotnet run` to try this latest version.

Before you start the next section, let's move the current code into a separate method. That makes it easier to start working with a new example. Place all the code you've written in a new method called `WorkWithStrings()`. Call that method at the top of your program. When you finish, your code should look like this:

```

using System;
using System.Collections.Generic;

WorkWithString();

void WorkWithString()
{
    var names = new List<string> { "<name>", "Ana", "Felipe" };
    foreach (var name in names)
    {
        Console.WriteLine($"Hello {name.ToUpper()}!");
    }

    Console.WriteLine();
    names.Add("Maria");
    names.Add("Bill");
    names.Remove("Ana");
    foreach (var name in names)
    {
        Console.WriteLine($"Hello {name.ToUpper()}!");
    }

    Console.WriteLine($"My name is {names[0]}");
    Console.WriteLine($"I've added {names[2]} and {names[3]} to the list");

    Console.WriteLine($"The list has {names.Count} people in it");

    var index = names.IndexOf("Felipe");
    if (index == -1)
    {
        Console.WriteLine($"When an item is not found, IndexOf returns {index}");
    }
    else
    {
        Console.WriteLine($"The name {names[index]} is at index {index}");
    }

    index = names.IndexOf("Not Found");
    if (index == -1)
    {
        Console.WriteLine($"When an item is not found, IndexOf returns {index}");
    }
    else
    {
        Console.WriteLine($"The name {names[index]} is at index {index}");
    }

    names.Sort();
    foreach (var name in names)
    {
        Console.WriteLine($"Hello {name.ToUpper()}!");
    }
}

```

Lists of other types

You've been using the `string` type in lists so far. Let's make a `List<T>` using a different type. Let's build a set of numbers.

Add the following to your program after you call `WorkWithStrings()` :

```
var fibonacciNumbers = new List<int> {1, 1};
```

That creates a list of integers, and sets the first two integers to the value 1. These are the first two values of a *Fibonacci Sequence*, a sequence of numbers. Each next Fibonacci number is found by taking the sum of the previous two numbers. Add this code:

```
var previous = fibonacciNumbers[fibonacciNumbers.Count - 1];
var previous2 = fibonacciNumbers[fibonacciNumbers.Count - 2];

fibonacciNumbers.Add(previous + previous2);

foreach (var item in fibonacciNumbers)
    Console.WriteLine(item);
```

Save the file and type `dotnet run` to see the results.

TIP

To concentrate on just this section, you can comment out the code that calls `WorkingWithStrings();`. Just put two `/` characters in front of the call like this: `// WorkingWithStrings();`.

Challenge

See if you can put together some of the concepts from this and earlier lessons. Expand on what you've built so far with Fibonacci Numbers. Try to write the code to generate the first 20 numbers in the sequence. (As a hint, the 20th Fibonacci number is 6765.)

Complete challenge

You can see an example solution by [looking at the finished sample code on GitHub](#).

With each iteration of the loop, you're taking the last two integers in the list, summing them, and adding that value to the list. The loop repeats until you've added 20 items to the list.

Congratulations, you've completed the list tutorial. You can continue with [additional](#) tutorials in your own development environment.

You can learn more about working with the `List` type in the .NET fundamentals article on [collections](#). You'll also learn about many other collection types.

General Structure of a C# Program

12/28/2021 • 2 minutes to read • [Edit Online](#)

C# programs consist of one or more files. Each file contains zero or more namespaces. A namespace contains types such as classes, structs, interfaces, enumerations, and delegates, or other namespaces. The following example is the skeleton of a C# program that contains all of these elements.

```
// A skeleton of a C# program
using System;

// Your program starts here:
Console.WriteLine("Hello world!");

namespace YourNamespace
{
    class YourClass
    {
    }

    struct YourStruct
    {
    }

    interface IYourInterface
    {
    }

    delegate int YourDelegate();

    enum YourEnum
    {
    }

    namespace YourNestedNamespace
    {
        struct YourStruct
        {
        }
    }
}
```

The preceding example uses *top-level statements* for the program's entry point. This feature was added in C# 9. Prior to C# 9, the entry point was a static method named `Main`, as shown in the following example:


```
// A skeleton of a C# program
using System;
namespace YourNamespace
{
    class YourClass
    {
    }

    struct YourStruct
    {
    }

    interface IYourInterface
    {
    }

    delegate int YourDelegate();

    enum YourEnum
    {
    }

    namespace YourNestedNamespace
    {
        struct YourStruct
        {
        }
    }

    class Program
    {
        static void Main(string[] args)
        {
            //Your program starts here...
            Console.WriteLine("Hello world!");
        }
    }
}
```

Related Sections

You learn about these program elements in the [types](#) section of the fundamentals guide:

- [Classes](#)
- [Structs](#)
- [Namespaces](#)
- [Interfaces](#)
- [Enums](#)
- [Delegates](#)

C# Language Specification

For more information, see [Basic concepts](#) in the [C# Language Specification](#). The language specification is the definitive source for C# syntax and usage.

Main() and command-line arguments

12/28/2021 • 9 minutes to read • [Edit Online](#)

The `Main` method is the entry point of a C# application. (Libraries and services do not require a `Main` method as an entry point.) When the application is started, the `Main` method is the first method that is invoked.

There can only be one entry point in a C# program. If you have more than one class that has a `Main` method, you must compile your program with the `StartupObject` compiler option to specify which `Main` method to use as the entry point. For more information, see [StartupObject \(C# Compiler Options\)](#).

```
using System;

class TestClass
{
    static void Main(string[] args)
    {
        // Display the number of command line arguments.
        Console.WriteLine(args.Length);
    }
}
```

Starting in C# 9, you can omit the `Main` method, and write C# statements as if they were in the `Main` method, as in the following example:

```
using System.Text;

StringBuilder builder = new();
builder.AppendLine("Hello");
builder.AppendLine("World!");

Console.WriteLine(builder.ToString());
```

For information about how to write application code with an implicit entry point method, see [Top-level statements](#).

Overview

- The `Main` method is the entry point of an executable program; it is where the program control starts and ends.
- `Main` is declared inside a class or struct. `Main` must be `static` and it need not be `public`. (In the earlier example, it receives the default access of `private`.) The enclosing class or struct is not required to be static.
- `Main` can either have a `void`, `int`, or, starting with C# 7.1, `Task`, or `Task<int>` return type.
- If and only if `Main` returns a `Task` or `Task<int>`, the declaration of `Main` may include the `async` modifier. This specifically excludes an `async void Main` method.
- The `Main` method can be declared with or without a `string[]` parameter that contains command-line arguments. When using Visual Studio to create Windows applications, you can add the parameter manually or else use the `GetCommandLineArgs()` method to obtain the command-line arguments. Parameters are read as zero-indexed command-line arguments. Unlike C and C++, the name of the program is not treated as the first command-line argument in the `args` array, but it is the first element of the `GetCommandLineArgs()` method.

The following list shows valid `Main` signatures:

```
public static void Main() { }
public static int Main() { }
public static void Main(string[] args) { }
public static int Main(string[] args) { }
public static async Task Main() { }
public static async Task<int> Main() { }
public static async Task Main(string[] args) { }
public static async Task<int> Main(string[] args) { }
```

The preceding examples all use the `public` accessor modifier. That's typical, but not required.

The addition of `async` and `Task`, `Task<int>` return types simplifies program code when console applications need to start and `await` asynchronous operations in `Main`.

Main() return values

You can return an `int` from the `Main` method by defining the method in one of the following ways:

<code>Main</code> METHOD CODE	<code>Main</code> SIGNATURE
No use of <code>args</code> or <code>await</code>	<code>static int Main()</code>
Uses <code>args</code> , no use of <code>await</code>	<code>static int Main(string[] args)</code>
No use of <code>args</code> , uses <code>await</code>	<code>static async Task<int> Main()</code>
Uses <code>args</code> and <code>await</code>	<code>static async Task<int> Main(string[] args)</code>

If the return value from `Main` is not used, returning `void` or `Task` allows for slightly simpler code.

<code>Main</code> METHOD CODE	<code>Main</code> SIGNATURE
No use of <code>args</code> or <code>await</code>	<code>static void Main()</code>
Uses <code>args</code> , no use of <code>await</code>	<code>static void Main(string[] args)</code>
No use of <code>args</code> , uses <code>await</code>	<code>static async Task Main()</code>
Uses <code>args</code> and <code>await</code>	<code>static async Task Main(string[] args)</code>

However, returning `int` or `Task<int>` enables the program to communicate status information to other programs or scripts that invoke the executable file.

The following example shows how the exit code for the process can be accessed.

This example uses [.NET Core](#) command-line tools. If you are unfamiliar with .NET Core command-line tools, you can learn about them in this [get-started article](#).

Create a new application by running `dotnet new console`. Modify the `Main` method in *Program.cs* as follows:

```
// Save this program as MainRetValTest.cs.
class MainRetValTest
{
    static int Main()
    {
        //...
        return 0;
    }
}
```

When a program is executed in Windows, any value returned from the `Main` function is stored in an environment variable. This environment variable can be retrieved using `ERRORLEVEL` from a batch file, or `$LastExitCode` from PowerShell.

You can build the application using the [dotnet CLI](#) `dotnet build` command.

Next, create a PowerShell script to run the application and display the result. Paste the following code into a text file and save it as `test.ps1` in the folder that contains the project. Run the PowerShell script by typing `test.ps1` at the PowerShell prompt.

Because the code returns zero, the batch file will report success. However, if you change `MainRetValTest.cs` to return a non-zero value and then recompile the program, subsequent execution of the PowerShell script will report failure.

```
dotnet run
if ($LastExitCode -eq 0) {
    Write-Host "Execution succeeded"
} else
{
    Write-Host "Execution Failed"
}
Write-Host "Return value = " $LastExitCode
```

```
Execution succeeded
Return value = 0
```

Async Main return values

When you declare an `async` return value for `Main`, the compiler generates the boilerplate code for calling asynchronous methods in `Main`. If you don't specify the `async` keyword, you need to write that code yourself, as shown in the following example. The code in the example ensures that your program runs until the asynchronous operation is completed:

```
public static void Main()
{
    AsyncConsoleWork().GetAwaiter().GetResult();
}

private static async Task<int> AsyncConsoleWork()
{
    // Main body here
    return 0;
}
```

This boilerplate code can be replaced by:

```
static async Task<int> Main(string[] args)
{
    return await AsyncConsoleWork();
}
```

An advantage of declaring `Main` as `async` is that the compiler always generates the correct code.

When the application entry point returns a `Task` or `Task<int>`, the compiler generates a new entry point that calls the entry point method declared in the application code. Assuming that this entry point is called `$GeneratedMain`, the compiler generates the following code for these entry points:

- `static Task Main()` results in the compiler emitting the equivalent of `private static void $GeneratedMain() => Main().GetAwaiter().GetResult();`
- `static Task Main(string[] args)` results in the compiler emitting the equivalent of `private static void $GeneratedMain(string[] args) => Main(args).GetAwaiter().GetResult();`
- `static Task<int> Main()` results in the compiler emitting the equivalent of `private static int $GeneratedMain() => Main().GetAwaiter().GetResult();`
- `static Task<int> Main(string[] args)` results in the compiler emitting the equivalent of `private static int $GeneratedMain(string[] args) => Main(args).GetAwaiter().GetResult();`

NOTE

If the examples used `async` modifier on the `Main` method, the compiler would generate the same code.

Command-Line Arguments

You can send arguments to the `Main` method by defining the method in one of the following ways:

MAIN METHOD CODE	MAIN SIGNATURE
No return value, no use of <code>await</code>	<code>static void Main(string[] args)</code>
Return value, no use of <code>await</code>	<code>static int Main(string[] args)</code>
No return value, uses <code>await</code>	<code>static async Task Main(string[] args)</code>
Return value, uses <code>await</code>	<code>static async Task<int> Main(string[] args)</code>

If the arguments are not used, you can omit `args` from the method signature for slightly simpler code:

MAIN METHOD CODE	MAIN SIGNATURE
No return value, no use of <code>await</code>	<code>static void Main()</code>
Return value, no use of <code>await</code>	<code>static int Main()</code>
No return value, uses <code>await</code>	<code>static async Task Main()</code>
Return value, uses <code>await</code>	<code>static async Task<int> Main()</code>

NOTE

You can also use [Environment.CommandLine](#) or [Environment.GetCommandLineArgs](#) to access the command-line arguments from any point in a console or Windows Forms application. To enable command-line arguments in the `Main` method signature in a Windows Forms application, you must manually modify the signature of `Main`. The code generated by the Windows Forms designer creates `Main` without an input parameter.

The parameter of the `Main` method is a [String](#) array that represents the command-line arguments. Usually you determine whether arguments exist by testing the `Length` property, for example:

```
if (args.Length == 0)
{
    System.Console.WriteLine("Please enter a numeric argument.");
    return 1;
}
```

TIP

The `args` array can't be null. So, it's safe to access the `Length` property without null checking.

You can also convert the string arguments to numeric types by using the [Convert](#) class or the `Parse` method. For example, the following statement converts the `string` to a `long` number by using the `Parse` method:

```
long num = Int64.Parse(args[0]);
```

It is also possible to use the C# type `long`, which aliases `Int64`:

```
long num = long.Parse(args[0]);
```

You can also use the `Convert` class method `ToInt64` to do the same thing:

```
long num = Convert.ToInt64(s);
```

For more information, see [Parse](#) and [Convert](#).

The following example shows how to use command-line arguments in a console application. The application takes one argument at run time, converts the argument to an integer, and calculates the factorial of the number. If no arguments are supplied, the application issues a message that explains the correct usage of the program.

To compile and run the application from a command prompt, follow these steps:

1. Paste the following code into any text editor, and then save the file as a text file with the name *Factorial.cs*.

```

using System;

public class Functions
{
    public static long Factorial(int n)
    {
        // Test for invalid input.
        if ((n < 0) || (n > 20))
        {
            return -1;
        }

        // Calculate the factorial iteratively rather than recursively.
        long tempResult = 1;
        for (int i = 1; i <= n; i++)
        {
            tempResult *= i;
        }
        return tempResult;
    }
}

class MainClass
{
    static int Main(string[] args)
    {
        // Test if input arguments were supplied.
        if (args.Length == 0)
        {
            Console.WriteLine("Please enter a numeric argument.");
            Console.WriteLine("Usage: Factorial <num>");
            return 1;
        }

        // Try to convert the input arguments to numbers. This will throw
        // an exception if the argument is not a number.
        // num = int.Parse(args[0]);
        int num;
        bool test = int.TryParse(args[0], out num);
        if (!test)
        {
            Console.WriteLine("Please enter a numeric argument.");
            Console.WriteLine("Usage: Factorial <num>");
            return 1;
        }

        // Calculate factorial.
        long result = Functions.Factorial(num);

        // Print result.
        if (result == -1)
            Console.WriteLine("Input must be >= 0 and <= 20.");
        else
            Console.WriteLine($"The Factorial of {num} is {result}.");

        return 0;
    }
}

// If 3 is entered on command line, the
// output reads: The factorial of 3 is 6.

```

2. From the **Start** screen or **Start** menu, open a Visual Studio **Developer Command Prompt** window, and then navigate to the folder that contains the file that you created.
3. Enter the following command to compile the application.

```
dotnet build
```

If your application has no compilation errors, an executable file that's named *Factorial.exe* is created.

4. Enter the following command to calculate the factorial of 3:

```
dotnet run -- 3
```

5. The command produces this output: `The factorial of 3 is 6.`

NOTE

When running an application in Visual Studio, you can specify command-line arguments in the [Debug Page, Project Designer](#).

C# language specification

For more information, see the [C# Language Specification](#). The language specification is the definitive source for C# syntax and usage.

See also

- [System.Environment](#)
- [How to display command line arguments](#)

Top-level statements – programs without `Main` methods

12/28/2021 • 2 minutes to read • [Edit Online](#)

Starting in C# 9, you don't have to explicitly include a `Main` method in a console application project. Instead, you can use the *top-level statements* feature to minimize the code you have to write. In this case, the compiler generates a class and `Main` method entry point for the application.

Here's a *Program.cs* file that is a complete C# program in C# 10:

```
Console.WriteLine("Hello World!");
```

Top-level statements let you write simple programs for small utilities such as Azure Functions and GitHub Actions. They also make it simpler for new C# programmers to get started learning and writing code.

The following sections explain the rules on what you can and can't do with top-level statements.

Only one top-level file

An application must have only one entry point. A project can have only one file with top-level statements. Putting top-level statements in more than one file in a project results in the following compiler error:

```
CS8802 Only one compilation unit can have top-level statements.
```

A project can have any number of additional source code files that don't have top-level statements.

No other entry points

You can write a `Main` method explicitly, but it can't function as an entry point. The compiler issues the following warning:

```
CS7022 The entry point of the program is global code; ignoring 'Main()' entry point.
```

In a project with top-level statements, you can't use the `-main` compiler option to select the entry point, even if the project has one or more `Main` methods.

`using` directives

If you include using directives, they must come first in the file, as in this example:

```
using System.Text;

StringBuilder builder = new();
builder.AppendLine("Hello");
builder.AppendLine("World!");

Console.WriteLine(builder.ToString());
```

Global namespace

Top-level statements are implicitly in the global namespace.

Namespaces and type definitions

A file with top-level statements can also contain namespaces and type definitions, but they must come after the top-level statements. For example:

```
MyClass.TestMethod();
MyNamespace.MyClass.MyMethod();

public class MyClass
{
    public static void TestMethod()
    {
        Console.WriteLine("Hello World!");
    }
}

namespace MyNamespace
{
    class MyClass
    {
        public static void MyMethod()
        {
            Console.WriteLine("Hello World from MyNamespace.MyClass.MyMethod!");
        }
    }
}
```

args

Top-level statements can reference the `args` variable to access any command-line arguments that were entered. The `args` variable is never null but its `Length` is zero if no command-line arguments were provided. For example:

```
if (args.Length > 0)
{
    foreach (var arg in args)
    {
        Console.WriteLine($"Argument={arg}");
    }
}
else
{
    Console.WriteLine("No arguments");
}
```

await

You can call an async method by using `await`. For example:

```
Console.Write("Hello ");
await Task.Delay(5000);
Console.WriteLine("World!");
```

Exit code for the process

To return an `int` value when the application ends, use the `return` statement as you would in a `Main` method that returns an `int`. For example:

```
string? s = Console.ReadLine();

int returnValue = int.Parse(s ?? "-1");
return returnValue;
```

Implicit entry point method

The compiler generates a method to serve as the program entry point for a project with top-level statements. The name of this method isn't actually `Main`, it's an implementation detail that your code can't reference directly. The signature of the method depends on whether the top-level statements contain the `await` keyword or the `return` statement. The following table shows what the method signature would look like, using the method name `Main` in the table for convenience.

TOP-LEVEL CODE CONTAINS	IMPLICIT <code>Main</code> SIGNATURE
<code>await</code> and <code>return</code>	<code>static async Task<int> Main(string[] args)</code>
<code>await</code>	<code>static async Task Main(string[] args)</code>
<code>return</code>	<code>static int Main(string[] args)</code>
No <code>await</code> or <code>return</code>	<code>static void Main(string[] args)</code>

C# language specification

For more information, see the [C# Language Specification](#). The language specification is the definitive source for C# syntax and usage.

[Feature specification - Top-level statements](#)

The C# type system

12/28/2021 • 14 minutes to read • [Edit Online](#)

C# is a strongly typed language. Every variable and constant has a type, as does every expression that evaluates to a value. Every method declaration specifies a name, the type and kind (value, reference, or output) for each input parameter and for the return value. The .NET class library defines built-in numeric types and complex types that represent a wide variety of constructs. These include the file system, network connections, collections and arrays of objects, and dates. A typical C# program uses types from the class library and user-defined types that model the concepts that are specific to the program's problem domain.

The information stored in a type can include the following items:

- The storage space that a variable of the type requires.
- The maximum and minimum values that it can represent.
- The members (methods, fields, events, and so on) that it contains.
- The base type it inherits from.
- The interface(s) it implements.
- The kinds of operations that are permitted.

The compiler uses type information to make sure all operations that are performed in your code are *type safe*. For example, if you declare a variable of type `int`, the compiler allows you to use the variable in addition and subtraction operations. If you try to perform those same operations on a variable of type `bool`, the compiler generates an error, as shown in the following example:

```
int a = 5;
int b = a + 2; //OK

bool test = true;

// Error. Operator '+' cannot be applied to operands of type 'int' and 'bool'.
int c = a + test;
```

NOTE

C and C++ developers, notice that in C#, `bool` is not convertible to `int`.

The compiler embeds the type information into the executable file as metadata. The common language runtime (CLR) uses that metadata at run time to further guarantee type safety when it allocates and reclaims memory.

Specifying types in variable declarations

When you declare a variable or constant in a program, you must either specify its type or use the `var` keyword to let the compiler infer the type. The following example shows some variable declarations that use both built-in numeric types and complex user-defined types:

```
// Declaration only:
float temperature;
string name;
MyClass myClass;

// Declaration with initializers (four examples):
char firstLetter = 'C';
var limit = 3;
int[] source = { 0, 1, 2, 3, 4, 5 };
var query = from item in source
            where item <= limit
            select item;
```

The types of method parameters and return values are specified in the method declaration. The following signature shows a method that requires an `int` as an input argument and returns a string:

```
public string GetName(int ID)
{
    if (ID < names.Length)
        return names[ID];
    else
        return String.Empty;
}
private string[] names = { "Spencer", "Sally", "Doug" };
```

After you declare a variable, you can't redeclare it with a new type, and you can't assign a value not compatible with its declared type. For example, you can't declare an `int` and then assign it a Boolean value of `true`. However, values can be converted to other types, for example when they're assigned to new variables or passed as method arguments. A *type conversion* that doesn't cause data loss is performed automatically by the compiler. A conversion that might cause data loss requires a *cast* in the source code.

For more information, see [Casting and Type Conversions](#).

Built-in types

C# provides a standard set of built-in types. These represent integers, floating point values, Boolean expressions, text characters, decimal values, and other types of data. There are also built-in `string` and `object` types. These types are available for you to use in any C# program. For the complete list of the built-in types, see [Built-in types](#).

Custom types

You use the `struct`, `class`, `interface`, `enum`, and `record` constructs to create your own custom types. The .NET class library itself is a collection of custom types that you can use in your own applications. By default, the most frequently used types in the class library are available in any C# program. Others become available only when you explicitly add a project reference to the assembly that defines them. After the compiler has a reference to the assembly, you can declare variables (and constants) of the types declared in that assembly in source code. For more information, see [.NET Class Library](#).

The common type system

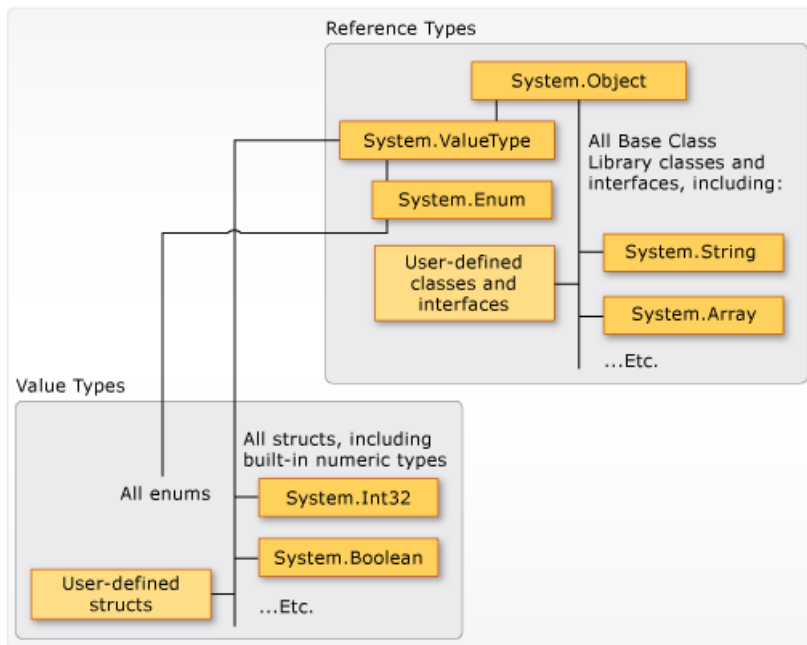
It's important to understand two fundamental points about the type system in .NET:

- It supports the principle of inheritance. Types can derive from other types, called *base types*. The derived type inherits (with some restrictions) the methods, properties, and other members of the base type. The base type can in turn derive from some other type, in which case the derived type inherits the members of both base types in its inheritance hierarchy. All types, including built-in numeric types such as `System.Int32` (C#

keyword: `int`), derive ultimately from a single base type, which is `System.Object` (C# keyword: `object`). This unified type hierarchy is called the **Common Type System** (CTS). For more information about inheritance in C#, see [Inheritance](#).

- Each type in the CTS is defined as either a *value type* or a *reference type*. These types include all custom types in the .NET class library and also your own user-defined types. Types that you define by using the `struct` keyword are value types; all the built-in numeric types are `structs` . Types that you define by using the `class` or `record` keyword are reference types. Reference types and value types have different compile-time rules, and different run-time behavior.

The following illustration shows the relationship between value types and reference types in the CTS.



NOTE

You can see that the most commonly used types are all organized in the `System` namespace. However, the namespace in which a type is contained has no relation to whether it is a value type or reference type.

Classes and structs are two of the basic constructs of the common type system in .NET. C# 9 adds records, which are a kind of class. Each is essentially a data structure that encapsulates a set of data and behaviors that belong together as a logical unit. The data and behaviors are the *members* of the class, struct, or record. The members include its methods, properties, events, and so on, as listed later in this article.

A class, struct, or record declaration is like a blueprint that is used to create instances or objects at run time. If you define a class, struct, or record named `Person`, `Person` is the name of the type. If you declare and initialize a variable `p` of type `Person`, `p` is said to be an object or instance of `Person`. Multiple instances of the same `Person` type can be created, and each instance can have different values in its properties and fields.

A class is a reference type. When an object of the type is created, the variable to which the object is assigned holds only a reference to that memory. When the object reference is assigned to a new variable, the new variable refers to the original object. Changes made through one variable are reflected in the other variable because they both refer to the same data.

A struct is a value type. When a struct is created, the variable to which the struct is assigned holds the struct's actual data. When the struct is assigned to a new variable, it's copied. The new variable and the original variable therefore contain two separate copies of the same data. Changes made to one copy don't affect the other copy.

Record types may be either reference types (`record class`) or value types (`record struct`).

In general, classes are used to model more complex behavior. Classes typically store data that is intended to be modified after a class object is created. Structs are best suited for small data structures. Structs typically store data that isn't intended to be modified after the struct is created. Record types are data structures with additional compiler synthesized members. Records typically store data that isn't intended to be modified after the object is created.

Value types

Value types derive from [System.ValueType](#), which derives from [System.Object](#). Types that derive from [System.ValueType](#) have special behavior in the CLR. Value type variables directly contain their values. The memory for a struct is allocated inline in whatever context the variable is declared. There's no separate heap allocation or garbage collection overhead for value-type variables. You can declare `record struct` types that are value types and include the synthesized members for [records](#).

There are two categories of value types: `struct` and `enum`.

The built-in numeric types are structs, and they have fields and methods that you can access:

```
// constant field on type byte.  
byte b = byte.MaxValue;
```

But you declare and assign values to them as if they're simple non-aggregate types:

```
byte num = 0xA;  
int i = 5;  
char c = 'Z';
```

Value types are *sealed*. You can't derive a type from any value type, for example [System.Int32](#). You can't define a struct to inherit from any user-defined class or struct because a struct can only inherit from [System.ValueType](#). However, a struct can implement one or more interfaces. You can cast a struct type to any interface type that it implements. This cast causes a *boxing* operation to wrap the struct inside a reference type object on the managed heap. Boxing operations occur when you pass a value type to a method that takes a [System.Object](#) or any interface type as an input parameter. For more information, see [Boxing and Unboxing](#).

You use the `struct` keyword to create your own custom value types. Typically, a struct is used as a container for a small set of related variables, as shown in the following example:

```
public struct Coords  
{  
    public int x, y;  
  
    public Coords(int p1, int p2)  
    {  
        x = p1;  
        y = p2;  
    }  
}
```

For more information about structs, see [Structure types](#). For more information about value types, see [Value types](#).

The other category of value types is `enum`. An enum defines a set of named integral constants. For example, the [System.IO.FileMode](#) enumeration in the .NET class library contains a set of named constant integers that specify how a file should be opened. It's defined as shown in the following example:

```
public enum FileMode
{
    CreateNew = 1,
    Create = 2,
    Open = 3,
    OpenOrCreate = 4,
    Truncate = 5,
    Append = 6,
}
```

The [System.IO.FileMode.Create](#) constant has a value of 2. However, the name is much more meaningful for humans reading the source code, and for that reason it's better to use enumerations instead of constant literal numbers. For more information, see [System.IO.FileMode](#).

All enums inherit from [System.Enum](#), which inherits from [System.ValueType](#). All the rules that apply to structs also apply to enums. For more information about enums, see [Enumeration types](#).

Reference types

A type that is defined as a `class`, `record`, `delegate`, array, or `interface` is a `reference type`.

When declaring a variable of a `reference type`, it contains the value `null` until you assign it with an instance of that type or create one using the `new` operator. Creation and assignment of a class are demonstrated in the following example:

```
MyClass myClass = new MyClass();
MyClass myClass2 = myClass;
```

An `interface` cannot be directly instantiated using the `new` operator. Instead, create and assign an instance of a class that implements the interface. Consider the following example:

```
MyClass myClass = new MyClass();

// Declare and assign using an existing value.
IMyInterface myInterface = myClass;

// Or create and assign a value in a single statement.
IMyInterface myInterface2 = new MyClass();
```

When the object is created, the memory is allocated on the managed heap. The variable holds only a reference to the location of the object. Types on the managed heap require overhead both when they're allocated and when they're reclaimed. *Garbage collection* is the automatic memory management functionality of the CLR, which performs the reclamation. However, garbage collection is also highly optimized, and in most scenarios it doesn't create a performance issue. For more information about garbage collection, see [Automatic Memory Management](#).

All arrays are reference types, even if their elements are value types. Arrays implicitly derive from the [System.Array](#) class. You declare and use them with the simplified syntax that is provided by C#, as shown in the following example:

```
// Declare and initialize an array of integers.
int[] nums = { 1, 2, 3, 4, 5 };

// Access an instance property of System.Array.
int len = nums.Length;
```

Reference types fully support inheritance. When you create a class, you can inherit from any other interface or

class that isn't defined as [sealed](#). Other classes can inherit from your class and override your virtual methods. For more information about how to create your own classes, see [Classes, structs, and records](#). For more information about inheritance and virtual methods, see [Inheritance](#).

Types of literal values

In C#, literal values receive a type from the compiler. You can specify how a numeric literal should be typed by appending a letter to the end of the number. For example, to specify that the value `4.56` should be treated as a `float`, append an "f" or "F" after the number: `4.56f`. If no letter is appended, the compiler will infer a type for the literal. For more information about which types can be specified with letter suffixes, see [Integral numeric types](#) and [Floating-point numeric types](#).

Because literals are typed, and all types derive ultimately from [System.Object](#), you can write and compile code such as the following code:

```
string s = "The answer is " + 5.ToString();
// Outputs: "The answer is 5"
Console.WriteLine(s);

Type type = 12345.GetType();
// Outputs: "System.Int32"
Console.WriteLine(type);
```

Generic types

A type can be declared with one or more *type parameters* that serve as a placeholder for the actual type (the *concrete type*). Client code provides the concrete type when it creates an instance of the type. Such types are called *generic types*. For example, the .NET type [System.Collections.Generic.List<T>](#) has one type parameter that by convention is given the name `T`. When you create an instance of the type, you specify the type of the objects that the list will contain, for example, `string`:

```
List<string> stringList = new List<string>();
stringList.Add("String example");
// compile time error adding a type other than a string:
stringList.Add(4);
```

The use of the type parameter makes it possible to reuse the same class to hold any type of element, without having to convert each element to [object](#). Generic collection classes are called *strongly typed collections* because the compiler knows the specific type of the collection's elements and can raise an error at compile time if, for example, you try to add an integer to the `stringList` object in the previous example. For more information, see [Generics](#).

Implicit types, anonymous types, and nullable value types

You can implicitly type a local variable (but not class members) by using the `var` keyword. The variable still receives a type at compile time, but the type is provided by the compiler. For more information, see [Implicitly Typed Local Variables](#).

It can be inconvenient to create a named type for simple sets of related values that you don't intend to store or pass outside method boundaries. You can create *anonymous types* for this purpose. For more information, see [Anonymous Types](#).

Ordinary value types can't have a value of `null`. However, you can create *nullable value types* by appending a `?` after the type. For example, `int?` is an `int` type that can also have the value `null`. Nullable value types are

instances of the generic struct type [System.Nullable<T>](#). Nullable value types are especially useful when you're passing data to and from databases in which numeric values might be `null`. For more information, see [Nullable value types](#).

Compile-time type and run-time type

A variable can have different compile-time and run-time types. The *compile-time type* is the declared or inferred type of the variable in the source code. The *run-time type* is the type of the instance referred to by that variable. Often those two types are the same, as in the following example:

```
string message = "This is a string of characters";
```

In other cases, the compile-time type is different, as shown in the following two examples:

```
object anotherMessage = "This is another string of characters";  
IEnumerable<char> someCharacters = "abcdefghijklmnopqrstuvwxyz";
```

In both of the preceding examples, the run-time type is a `string`. The compile-time type is `object` in the first line, and `IEnumerable<char>` in the second.

If the two types are different for a variable, it's important to understand when the compile-time type and the run-time type apply. The compile-time type determines all the actions taken by the compiler. These compiler actions include method call resolution, overload resolution, and available implicit and explicit casts. The run-time type determines all actions that are resolved at run time. These run-time actions include dispatching virtual method calls, evaluating `is` and `switch` expressions, and other type testing APIs. To better understand how your code interacts with types, recognize which action applies to which type.

Related sections

For more information, see the following articles:

- [Builtin types](#)
- [Value Types](#)
- [Reference Types](#)

C# language specification

For more information, see the [C# Language Specification](#). The language specification is the definitive source for C# syntax and usage.

Declare namespaces to organize types

12/28/2021 • 2 minutes to read • [Edit Online](#)

Namespaces are heavily used in C# programming in two ways. First, .NET uses namespaces to organize its many classes, as follows:

```
System.Console.WriteLine("Hello World!");
```

`System` is a namespace and `Console` is a class in that namespace. The `using` keyword can be used so that the complete name isn't required, as in the following example:

```
using System;
```

```
Console.WriteLine("Hello World!");
```

For more information, see the [using Directive](#).

IMPORTANT

The C# templates for .NET 6 use *top level statements*. Your application may not match the code in this article, if you've already upgraded to the .NET 6 previews. For more information see the article on [New C# templates generate top level statements](#)

The .NET 6 SDK also adds a set of *implicit* `global using` directives for projects that use the following SDKs:

- Microsoft.NET.Sdk
- Microsoft.NET.Sdk.Web
- Microsoft.NET.Sdk.Worker

These implicit `global using` directives include the most common namespaces for the project type.

Second, declaring your own namespaces can help you control the scope of class and method names in larger programming projects. Use the `namespace` keyword to declare a namespace, as in the following example:

```
namespace SampleNamespace
{
    class SampleClass
    {
        public void SampleMethod()
        {
            System.Console.WriteLine(
                "SampleMethod inside SampleNamespace");
        }
    }
}
```

The name of the namespace must be a valid C# [identifier name](#).

Beginning with C# 10, you can declare a namespace for all types defined in that file, as shown in the following example:

```
namespace SampleNamespace;

class AnotherSampleClass
{
    public void AnotherSampleMethod()
    {
        System.Console.WriteLine(
            "SampleMethod inside SampleNamespace");
    }
}
```

The advantage of this new syntax is that it's simpler, saving horizontal space and braces. That makes your code easier to read.

Namespaces overview

Namespaces have the following properties:

- They organize large code projects.
- They're delimited by using the `.` operator.
- The `using` directive obviates the requirement to specify the name of the namespace for every class.
- The `global` namespace is the "root" namespace: `global::System` will always refer to the .NET [System](#) namespace.

C# language specification

For more information, see the [Namespaces](#) section of the [C# language specification](#).

Introduction to classes

12/28/2021 • 5 minutes to read • [Edit Online](#)

Reference types

A type that is defined as a `class` is a *reference type*. At run time, when you declare a variable of a reference type, the variable contains the value `null` until you explicitly create an instance of the class by using the `new` operator, or assign it an object of a compatible type that may have been created elsewhere, as shown in the following example:

```
//Declaring an object of type MyClass.  
MyClass mc = new MyClass();  
  
//Declaring another object of the same type, assigning it the value of the first object.  
MyClass mc2 = mc;
```

When the object is created, enough memory is allocated on the managed heap for that specific object, and the variable holds only a reference to the location of said object. Types on the managed heap require overhead both when they are allocated and when they are reclaimed by the automatic memory management functionality of the CLR, which is known as *garbage collection*. However, garbage collection is also highly optimized and in most scenarios, it does not create a performance issue. For more information about garbage collection, see [Automatic memory management and garbage collection](#).

Declaring Classes

Classes are declared by using the `class` keyword followed by a unique identifier, as shown in the following example:

```
//[access modifier] - [class] - [identifier]  
public class Customer  
{  
    // Fields, properties, methods and events go here...  
}
```

The `class` keyword is preceded by the access level. Because `public` is used in this case, anyone can create instances of this class. The name of the class follows the `class` keyword. The name of the class must be a valid C# [identifier name](#). The remainder of the definition is the class body, where the behavior and data are defined. Fields, properties, methods, and events on a class are collectively referred to as *class members*.

Creating objects

Although they are sometimes used interchangeably, a class and an object are different things. A class defines a type of object, but it is not an object itself. An object is a concrete entity based on a class, and is sometimes referred to as an instance of a class.

Objects can be created by using the `new` keyword followed by the name of the class that the object will be based on, like this:

```
Customer object1 = new Customer();
```

When an instance of a class is created, a reference to the object is passed back to the programmer. In the previous example, `object1` is a reference to an object that is based on `Customer`. This reference refers to the new object but does not contain the object data itself. In fact, you can create an object reference without creating an object at all:

```
Customer object2;
```

We don't recommend creating object references such as the preceding one that don't refer to an object because trying to access an object through such a reference will fail at run time. However, such a reference can be made to refer to an object, either by creating a new object, or by assigning it an existing object, such as this:

```
Customer object3 = new Customer();  
Customer object4 = object3;
```

This code creates two object references that both refer to the same object. Therefore, any changes to the object made through `object3` are reflected in subsequent uses of `object4`. Because objects that are based on classes are referred to by reference, classes are known as reference types.

Class inheritance

Classes fully support *inheritance*, a fundamental characteristic of object-oriented programming. When you create a class, you can inherit from any other class that is not defined as `sealed`, and other classes can inherit from your class and override class virtual methods. Furthermore, you can implement one or more interfaces.

Inheritance is accomplished by using a *derivation*, which means a class is declared by using a *base class* from which it inherits data and behavior. A base class is specified by appending a colon and the name of the base class following the derived class name, like this:

```
public class Manager : Employee  
{  
    // Employee fields, properties, methods and events are inherited  
    // New Manager fields, properties, methods and events go here...  
}
```

When a class declares a base class, it inherits all the members of the base class except the constructors. For more information, see [Inheritance](#).

A class in C# can only directly inherit from one base class. However, because a base class may itself inherit from another class, a class may indirectly inherit multiple base classes. Furthermore, a class can directly implement one or more interfaces. For more information, see [Interfaces](#).

A class can be declared `abstract`. An abstract class contains abstract methods that have a signature definition but no implementation. Abstract classes cannot be instantiated. They can only be used through derived classes that implement the abstract methods. By contrast, a `sealed` class does not allow other classes to derive from it. For more information, see [Abstract and Sealed Classes and Class Members](#).

Class definitions can be split between different source files. For more information, see [Partial Classes and Methods](#).

Example

The following example defines a public class that contains an [auto-implemented property](#), a method, and a special method called a constructor. For more information, see [Properties](#), [Methods](#), and [Constructors](#) articles. The instances of the class are then instantiated with the `new` keyword.

```

using System;

public class Person
{
    // Constructor that takes no arguments:
    public Person()
    {
        Name = "unknown";
    }

    // Constructor that takes one argument:
    public Person(string name)
    {
        Name = name;
    }

    // Auto-implemented readonly property:
    public string Name { get; }

    // Method that overrides the base class (System.Object) implementation.
    public override string ToString()
    {
        return Name;
    }
}

class TestPerson
{
    static void Main()
    {
        // Call the constructor that has no parameters.
        var person1 = new Person();
        Console.WriteLine(person1.Name);

        // Call the constructor that has one parameter.
        var person2 = new Person("Sarah Jones");
        Console.WriteLine(person2.Name);
        // Get the string representation of the person2 instance.
        Console.WriteLine(person2);
    }
}

// Output:
// unknown
// Sarah Jones
// Sarah Jones

```

C# Language Specification

For more information, see the [C# Language Specification](#). The language specification is the definitive source for C# syntax and usage.

Introduction to records

12/28/2021 • 3 minutes to read • [Edit Online](#)

A [record](#) is a [class](#) or [struct](#) that provides special syntax and behavior for working with data models.

When to use records

Consider using a record in place of a class or struct in the following scenarios:

- You want to define a data model that depends on [value equality](#).
- You want to define a type for which objects are immutable.

Value equality

For records, value equality means that two variables of a record type are equal if the types match and all property and field values match. For other reference types such as classes, equality means [reference equality](#). That is, two variables of a class type are equal if they refer to the same object. Methods and operators that determine equality of two record instances use value equality.

Not all data models work well with value equality. For example, [Entity Framework Core](#) depends on reference equality to ensure that it uses only one instance of an entity type for what is conceptually one entity. For this reason, record types aren't appropriate for use as entity types in Entity Framework Core.

Immutability

An immutable type is one that prevents you from changing any property or field values of an object after it's instantiated. Immutability can be useful when you need a type to be thread-safe or you're depending on a hash code remaining the same in a hash table. Records provide concise syntax for creating and working with immutable types.

Immutability isn't appropriate for all data scenarios. [Entity Framework Core](#), for example, doesn't support updating with immutable entity types.

How records differ from classes and structs

The same syntax that [declares](#) and [instantiates](#) classes or structs can be used with records. Just substitute the `class` keyword with the `record`, or use `record struct` instead of `struct`. Likewise, the same syntax for expressing inheritance relationships is supported by record classes. Records differ from classes in the following ways:

- You can use [positional parameters](#) to create and instantiate a type with immutable properties.
- The same methods and operators that indicate reference equality or inequality in classes (such as `Object.Equals(Object)` and `==`), indicate [value equality or inequality](#) in records.
- You can use a `with` [expression](#) to create a copy of an immutable object with new values in selected properties.
- A record's `ToString` method creates a formatted string that shows an object's type name and the names and values of all its public properties.
- A record can [inherit from another record](#). A record can't inherit from a class, and a class can't inherit from a record.

Record structs differ from structs in that the compiler synthesizes the methods for equality, and `ToString`. The compiler synthesizes a `Deconstruct` method for positional record structs.

Examples

The following example defines a public record that uses positional parameters to declare and instantiate a record. It then prints the type name and property values:

```
public record Person(string FirstName, string LastName);

public static void Main()
{
    Person person = new("Nancy", "Davolio");
    Console.WriteLine(person);
    // output: Person { FirstName = Nancy, LastName = Davolio }
}
```

The following example demonstrates value equality in records:

```
public record Person(string FirstName, string LastName, string[] PhoneNumbers);

public static void Main()
{
    var phoneNumbers = new string[2];
    Person person1 = new("Nancy", "Davolio", phoneNumbers);
    Person person2 = new("Nancy", "Davolio", phoneNumbers);
    Console.WriteLine(person1 == person2); // output: True

    person1.PhoneNumbers[0] = "555-1234";
    Console.WriteLine(person1 == person2); // output: True

    Console.WriteLine(ReferenceEquals(person1, person2)); // output: False
}
```

The following example demonstrates use of a `with` expression to copy an immutable object and change one of the properties:

```
public record Person(string FirstName, string LastName)
{
    public string[] PhoneNumbers { get; init; }
}

public static void Main()
{
    Person person1 = new("Nancy", "Davolio") { PhoneNumbers = new string[1] };
    Console.WriteLine(person1);
    // output: Person { FirstName = Nancy, LastName = Davolio, PhoneNumbers = System.String[] }

    Person person2 = person1 with { FirstName = "John" };
    Console.WriteLine(person2);
    // output: Person { FirstName = John, LastName = Davolio, PhoneNumbers = System.String[] }
    Console.WriteLine(person1 == person2); // output: False

    person2 = person1 with { PhoneNumbers = new string[1] };
    Console.WriteLine(person2);
    // output: Person { FirstName = Nancy, LastName = Davolio, PhoneNumbers = System.String[] }
    Console.WriteLine(person1 == person2); // output: False

    person2 = person1 with { };
    Console.WriteLine(person1 == person2); // output: True
}
```

For more information, see [Records \(C# reference\)](#).

C# Language Specification

For more information, see the [C# Language Specification](#). The language specification is the definitive source for C# syntax and usage.

Interfaces - define behavior for multiple types

12/28/2021 • 4 minutes to read • [Edit Online](#)

An interface contains definitions for a group of related functionalities that a non-abstract `class` or a `struct` must implement. An interface may define `static` methods, which must have an implementation. Beginning with C# 8.0, an interface may define a default implementation for members. An interface may not declare instance data such as fields, auto-implemented properties, or property-like events.

By using interfaces, you can, for example, include behavior from multiple sources in a class. That capability is important in C# because the language doesn't support multiple inheritance of classes. In addition, you must use an interface if you want to simulate inheritance for structs, because they can't actually inherit from another struct or class.

You define an interface by using the `interface` keyword as the following example shows.

```
interface IEquatable<T>
{
    bool Equals(T obj);
}
```

The name of an interface must be a valid C# [identifier name](#). By convention, interface names begin with a capital `I`.

Any class or struct that implements the `IEquatable<T>` interface must contain a definition for an `Equals` method that matches the signature that the interface specifies. As a result, you can count on a class that implements `IEquatable<T>` to contain an `Equals` method with which an instance of the class can determine whether it's equal to another instance of the same class.

The definition of `IEquatable<T>` doesn't provide an implementation for `Equals`. A class or struct can implement multiple interfaces, but a class can only inherit from a single class.

For more information about abstract classes, see [Abstract and Sealed Classes and Class Members](#).

Interfaces can contain instance methods, properties, events, indexers, or any combination of those four member types. Interfaces may contain static constructors, fields, constants, or operators. An interface can't contain instance fields, instance constructors, or finalizers. Interface members are public by default, and you can explicitly specify accessibility modifiers, such as `public`, `protected`, `internal`, `private`, `protected internal`, or `private protected`. A `private` member must have a default implementation.

To implement an interface member, the corresponding member of the implementing class must be public, non-static, and have the same name and signature as the interface member.

When a class or struct implements an interface, the class or struct must provide an implementation for all of the members that the interface declares but doesn't provide a default implementation for. However, if a base class implements an interface, any class that's derived from the base class inherits that implementation.

The following example shows an implementation of the `IEquatable<T>` interface. The implementing class, `Car`, must provide an implementation of the `Equals` method.

```

public class Car : IEquatable<Car>
{
    public string Make { get; set; }
    public string Model { get; set; }
    public string Year { get; set; }

    // Implementation of IEquatable<T> interface
    public bool Equals(Car car)
    {
        return (this.Make, this.Model, this.Year) ==
            (car.Make, car.Model, car.Year);
    }
}

```

Properties and indexers of a class can define extra accessors for a property or indexer that's defined in an interface. For example, an interface might declare a property that has a [get](#) accessor. The class that implements the interface can declare the same property with both a `get` and `set` accessor. However, if the property or indexer uses explicit implementation, the accessors must match. For more information about explicit implementation, see [Explicit Interface Implementation](#) and [Interface Properties](#).

Interfaces can inherit from one or more interfaces. The derived interface inherits the members from its base interfaces. A class that implements a derived interface must implement all members in the derived interface, including all members of the derived interface's base interfaces. That class may be implicitly converted to the derived interface or any of its base interfaces. A class might include an interface multiple times through base classes that it inherits or through interfaces that other interfaces inherit. However, the class can provide an implementation of an interface only one time and only if the class declares the interface as part of the definition of the class (`class ClassName : InterfaceName`). If the interface is inherited because you inherited a base class that implements the interface, the base class provides the implementation of the members of the interface. However, the derived class can reimplement any virtual interface members instead of using the inherited implementation. When interfaces declare a default implementation of a method, any class implementing that interface inherits that implementation (You need to cast the class instance to the interface type to access the default implementation on the Interface member).

A base class can also implement interface members by using virtual members. In that case, a derived class can change the interface behavior by overriding the virtual members. For more information about virtual members, see [Polymorphism](#).

Interfaces summary

An interface has the following properties:

- In C# versions earlier than 8.0, an interface is like an abstract base class with only abstract members. A class or struct that implements the interface must implement all its members.
- Beginning with C# 8.0, an interface may define default implementations for some or all of its members. A class or struct that implements the interface doesn't have to implement members that have default implementations. For more information, see [default interface methods](#).
- An interface can't be instantiated directly. Its members are implemented by any class or struct that implements the interface.
- A class or struct can implement multiple interfaces. A class can inherit a base class and also implement one or more interfaces.

Generic classes and methods

12/28/2021 • 3 minutes to read • [Edit Online](#)

Generics introduces the concept of type parameters to .NET, which make it possible to design classes and methods that defer the specification of one or more types until the class or method is declared and instantiated by client code. For example, by using a generic type parameter `T`, you can write a single class that other client code can use without incurring the cost or risk of runtime casts or boxing operations, as shown here:

```
// Declare the generic class.
public class GenericList<T>
{
    public void Add(T input) { }
}
class TestGenericList
{
    private class ExampleClass { }
    static void Main()
    {
        // Declare a list of type int.
        GenericList<int> list1 = new GenericList<int>();
        list1.Add(1);

        // Declare a list of type string.
        GenericList<string> list2 = new GenericList<string>();
        list2.Add("");

        // Declare a list of type ExampleClass.
        GenericList<ExampleClass> list3 = new GenericList<ExampleClass>();
        list3.Add(new ExampleClass());
    }
}
```

Generic classes and methods combine reusability, type safety, and efficiency in a way that their non-generic counterparts cannot. Generics are most frequently used with collections and the methods that operate on them. The [System.Collections.Generic](#) namespace contains several generic-based collection classes. The non-generic collections, such as [ArrayList](#) are not recommended and are maintained for compatibility purposes. For more information, see [Generics in .NET](#).

You can also create custom generic types and methods to provide your own generalized solutions and design patterns that are type-safe and efficient. The following code example shows a simple generic linked-list class for demonstration purposes. (In most cases, you should use the [List<T>](#) class provided by .NET instead of creating your own.) The type parameter `T` is used in several locations where a concrete type would ordinarily be used to indicate the type of the item stored in the list. It is used in the following ways:

- As the type of a method parameter in the `AddHead` method.
- As the return type of the `Data` property in the nested `Node` class.
- As the type of the private member `data` in the nested class.

`T` is available to the nested `Node` class. When `GenericList<T>` is instantiated with a concrete type, for example as a `GenericList<int>`, each occurrence of `T` will be replaced with `int`.

```

// type parameter T in angle brackets
public class GenericList<T>
{
    // The nested class is also generic on T.
    private class Node
    {
        // T used in non-generic constructor.
        public Node(T t)
        {
            next = null;
            data = t;
        }

        private Node next;
        public Node Next
        {
            get { return next; }
            set { next = value; }
        }

        // T as private member data type.
        private T data;

        // T as return type of property.
        public T Data
        {
            get { return data; }
            set { data = value; }
        }
    }

    private Node head;

    // constructor
    public GenericList()
    {
        head = null;
    }

    // T as method parameter type:
    public void AddHead(T t)
    {
        Node n = new Node(t);
        n.Next = head;
        head = n;
    }

    public IEnumerator<T> GetEnumerator()
    {
        Node current = head;

        while (current != null)
        {
            yield return current.Data;
            current = current.Next;
        }
    }
}

```

The following code example shows how client code uses the generic `GenericList<T>` class to create a list of integers. Simply by changing the type argument, the following code could easily be modified to create lists of strings or any other custom type:

```

class TestGenericList
{
    static void Main()
    {
        // int is the type argument
        GenericList<int> list = new GenericList<int>();

        for (int x = 0; x < 10; x++)
        {
            list.AddHead(x);
        }

        foreach (int i in list)
        {
            System.Console.Write(i + " ");
        }
        System.Console.WriteLine("\nDone");
    }
}

```

Generics overview

- Use generic types to maximize code reuse, type safety, and performance.
- The most common use of generics is to create collection classes.
- The .NET class library contains several generic collection classes in the [System.Collections.Generic](#) namespace. The generic collections should be used whenever possible instead of classes such as [ArrayList](#) in the [System.Collections](#) namespace.
- You can create your own generic interfaces, classes, methods, events, and delegates.
- Generic classes may be constrained to enable access to methods on particular data types.
- Information on the types that are used in a generic data type may be obtained at run-time by using reflection.

C# language specification

For more information, see the [C# Language Specification](#).

See also

- [System.Collections.Generic](#)
- [Generics in .NET](#)

Anonymous Types

12/28/2021 • 3 minutes to read • [Edit Online](#)

Anonymous types provide a convenient way to encapsulate a set of read-only properties into a single object without having to explicitly define a type first. The type name is generated by the compiler and is not available at the source code level. The type of each property is inferred by the compiler.

You create anonymous types by using the `new` operator together with an object initializer. For more information about object initializers, see [Object and Collection Initializers](#).

The following example shows an anonymous type that is initialized with two properties named `Amount` and `Message`.

```
var v = new { Amount = 108, Message = "Hello" };

// Rest the mouse pointer over v.Amount and v.Message in the following
// statement to verify that their inferred types are int and string.
Console.WriteLine(v.Amount + v.Message);
```

Anonymous types typically are used in the `select` clause of a query expression to return a subset of the properties from each object in the source sequence. For more information about queries, see [LINQ in C#](#).

Anonymous types contain one or more public read-only properties. No other kinds of class members, such as methods or events, are valid. The expression that is used to initialize a property cannot be `null`, an anonymous function, or a pointer type.

The most common scenario is to initialize an anonymous type with properties from another type. In the following example, assume that a class exists that is named `Product`. Class `Product` includes `Color` and `Price` properties, together with other properties that you are not interested in. Variable `products` is a collection of `Product` objects. The anonymous type declaration starts with the `new` keyword. The declaration initializes a new type that uses only two properties from `Product`. Using anonymous types causes a smaller amount of data to be returned in the query.

If you do not specify member names in the anonymous type, the compiler gives the anonymous type members the same name as the property being used to initialize them. You provide a name for a property that is being initialized with an expression, as shown in the previous example. In the following example, the names of the properties of the anonymous type are `Color` and `Price`.

```
var productQuery =
    from prod in products
    select new { prod.Color, prod.Price };

foreach (var v in productQuery)
{
    Console.WriteLine("Color={0}, Price={1}", v.Color, v.Price);
}
```

Typically, when you use an anonymous type to initialize a variable, you declare the variable as an implicitly typed local variable by using `var`. The type name cannot be specified in the variable declaration because only the compiler has access to the underlying name of the anonymous type. For more information about `var`, see [Implicitly Typed Local Variables](#).

You can create an array of anonymously typed elements by combining an implicitly typed local variable and an implicitly typed array, as shown in the following example.

```
var anonArray = new[] { new { name = "apple", diam = 4 }, new { name = "grape", diam = 1 } };
```

Anonymous types are `class` types that derive directly from `object`, and that cannot be cast to any type except `object`. The compiler provides a name for each anonymous type, although your application cannot access it. From the perspective of the common language runtime, an anonymous type is no different from any other reference type.

If two or more anonymous object initializers in an assembly specify a sequence of properties that are in the same order and that have the same names and types, the compiler treats the objects as instances of the same type. They share the same compiler-generated type information.

Anonymous types support non-destructive mutation in the form of [with expressions](#). This enables you to create a new instance of an anonymous type where one or more properties have new values:

```
var apple = new { Item = "apples", Price = 1.35 };
var onSale = apple with { Price = 0.79 };
Console.WriteLine(apple);
Console.WriteLine(onSale);
```

You cannot declare a field, a property, an event, or the return type of a method as having an anonymous type. Similarly, you cannot declare a formal parameter of a method, property, constructor, or indexer as having an anonymous type. To pass an anonymous type, or a collection that contains anonymous types, as an argument to a method, you can declare the parameter as type `object`. However, using `object` for anonymous types defeats the purpose of strong typing. If you must store query results or pass them outside the method boundary, consider using an ordinary named struct or class instead of an anonymous type.

Because the [Equals](#) and [GetHashCode](#) methods on anonymous types are defined in terms of the `Equals` and `GetHashCode` methods of the properties, two instances of the same anonymous type are equal only if all their properties are equal.

Object-oriented programming

12/28/2021 • 4 minutes to read • [Edit Online](#)

Encapsulation

Encapsulation is sometimes referred to as the first pillar or principle of object-oriented programming. A class or struct can specify how accessible each of its members is to code outside of the class or struct. Methods and variables that aren't intended to be used from outside of the class or assembly can be hidden to limit the potential for coding errors or malicious exploits. For more information, see [Object-oriented programming](#).

Members

The *members* of a type include all methods, fields, constants, properties, and events. In C#, there are no global variables or methods as there are in some other languages. Even a program's entry point, the `Main` method, must be declared within a class or struct (implicitly when you use [top-level statements](#)).

The following list includes all the various kinds of members that may be declared in a class, struct, or record.

- Fields
- Constants
- Properties
- Methods
- Constructors
- Events
- Finalizers
- Indexers
- Operators
- Nested Types

Accessibility

Some methods and properties are meant to be called or accessed from code outside a class or struct, known as *client code*. Other methods and properties might be only for use in the class or struct itself. It's important to limit the accessibility of your code so that only the intended client code can reach it. You specify how accessible your types and their members are to client code by using the following access modifiers:

- `public`
- `protected`
- `internal`
- `protected internal`
- `private`
- `private protected`.

The default accessibility is `private`.

Inheritance

Classes (but not structs) support the concept of inheritance. A class that derives from another class, called the *base class*, automatically contains all the public, protected, and internal members of the base class except its

constructors and finalizers. For more information, see [Inheritance](#) and [Polymorphism](#).

Classes may be declared as [abstract](#), which means that one or more of their methods have no implementation. Although abstract classes cannot be instantiated directly, they can serve as base classes for other classes that provide the missing implementation. Classes can also be declared as [sealed](#) to prevent other classes from inheriting from them.

Interfaces

Classes, structs, and records can implement multiple interfaces. To implement from an interface means that the type implements all the methods defined in the interface. For more information, see [Interfaces](#).

Generic Types

Classes, structs, and records can be defined with one or more type parameters. Client code supplies the type when it creates an instance of the type. For example, The [List<T>](#) class in the [System.Collections.Generic](#) namespace is defined with one type parameter. Client code creates an instance of a `List<string>` or `List<int>` to specify the type that the list will hold. For more information, see [Generics](#).

Static Types

Classes (but not structs or records) can be declared as `static`. A static class can contain only static members and can't be instantiated with the `new` keyword. One copy of the class is loaded into memory when the program loads, and its members are accessed through the class name. Classes, structs, and records can contain static members.

Nested Types

A class, struct, or record can be nested within another class, struct, or record.

Partial Types

You can define part of a class, struct, or method in one code file and another part in a separate code file.

Object Initializers

You can instantiate and initialize class or struct objects, and collections of objects, by assigning values to its properties.

Anonymous Types

In situations where it isn't convenient or necessary to create a named class you use anonymous types. Anonymous types are defined by their named data members.

Extension Methods

You can "extend" a class without creating a derived class by creating a separate type. That type contains methods that can be called as if they belonged to the original type.

Implicitly Typed Local Variables

Within a class or struct method, you can use implicit typing to instruct the compiler to determine a variable's type at compile time.

Records

C# 9 introduces the `record` type, a reference type that you can create instead of a class or a struct. Records are classes with built-in behavior for encapsulating data in immutable types. C# 10 introduces the `record struct` value type. A record (either `record class` or `record struct`) provides the following features:

- Concise syntax for creating a reference type with immutable properties.
- Value equality. Two variables of a record type are equal if they have the same type, and if, for every field, the values in both records are equal. Classes use reference equality: two variables of a class type are equal if they refer to the same object.
- Concise syntax for nondestructive mutation. A `with` expression lets you create a new record instance that is a copy of an existing instance but with specified property values changed.
- Built-in formatting for display. The `ToString` method prints the record type name and the names and values of public properties.
- Support for inheritance hierarchies in record classes. Record classes support inheritance. Record structs don't support inheritance.

For more information, see [Records](#).

C# Language Specification

For more information, see the [C# Language Specification](#). The language specification is the definitive source for C# syntax and usage.

Objects - create instances of types

12/28/2021 • 5 minutes to read • [Edit Online](#)

A class or struct definition is like a blueprint that specifies what the type can do. An object is basically a block of memory that has been allocated and configured according to the blueprint. A program may create many objects of the same class. Objects are also called instances, and they can be stored in either a named variable or in an array or collection. Client code is the code that uses these variables to call the methods and access the public properties of the object. In an object-oriented language such as C#, a typical program consists of multiple objects interacting dynamically.

NOTE

Static types behave differently than what is described here. For more information, see [Static Classes and Static Class Members](#).

Struct Instances vs. Class Instances

Because classes are reference types, a variable of a class object holds a reference to the address of the object on the managed heap. If a second variable of the same type is assigned to the first variable, then both variables refer to the object at that address. This point is discussed in more detail later in this article.

Instances of classes are created by using the `new` operator. In the following example, `Person` is the type and `person1` and `person2` are instances, or objects, of that type.

```

using System;

public class Person
{
    public string Name { get; set; }
    public int Age { get; set; }
    public Person(string name, int age)
    {
        Name = name;
        Age = age;
    }
    // Other properties, methods, events...
}

class Program
{
    static void Main()
    {
        Person person1 = new Person("Leopold", 6);
        Console.WriteLine("person1 Name = {0} Age = {1}", person1.Name, person1.Age);

        // Declare new person, assign person1 to it.
        Person person2 = person1;

        // Change the name of person2, and person1 also changes.
        person2.Name = "Molly";
        person2.Age = 16;

        Console.WriteLine("person2 Name = {0} Age = {1}", person2.Name, person2.Age);
        Console.WriteLine("person1 Name = {0} Age = {1}", person1.Name, person1.Age);
    }
}
/*
Output:
person1 Name = Leopold Age = 6
person2 Name = Molly Age = 16
person1 Name = Molly Age = 16
*/

```

Because structs are value types, a variable of a struct object holds a copy of the entire object. Instances of structs can also be created by using the `new` operator, but this isn't required, as shown in the following example:

```

using System;

namespace Example
{
    public struct Person
    {
        public string Name;
        public int Age;
        public Person(string name, int age)
        {
            Name = name;
            Age = age;
        }
    }

    public class Application
    {
        static void Main()
        {
            // Create struct instance and initialize by using "new".
            // Memory is allocated on thread stack.
            Person p1 = new Person("Alex", 9);
            Console.WriteLine("p1 Name = {0} Age = {1}", p1.Name, p1.Age);

            // Create new struct object. Note that struct can be initialized
            // without using "new".
            Person p2 = p1;

            // Assign values to p2 members.
            p2.Name = "Spencer";
            p2.Age = 7;
            Console.WriteLine("p2 Name = {0} Age = {1}", p2.Name, p2.Age);

            // p1 values remain unchanged because p2 is copy.
            Console.WriteLine("p1 Name = {0} Age = {1}", p1.Name, p1.Age);
        }
    }
    /*
    Output:
    p1 Name = Alex Age = 9
    p2 Name = Spencer Age = 7
    p1 Name = Alex Age = 9
    */
}

```

The memory for both `p1` and `p2` is allocated on the thread stack. That memory is reclaimed along with the type or method in which it's declared. This is one reason why structs are copied on assignment. By contrast, the memory that is allocated for a class instance is automatically reclaimed (garbage collected) by the common language runtime when all references to the object have gone out of scope. It isn't possible to deterministically destroy a class object like you can in C++. For more information about garbage collection in .NET, see [Garbage Collection](#).

NOTE

The allocation and deallocation of memory on the managed heap is highly optimized in the common language runtime. In most cases there is no significant difference in the performance cost of allocating a class instance on the heap versus allocating a struct instance on the stack.

Object Identity vs. Value Equality

When you compare two objects for equality, you must first distinguish whether you want to know whether the

two variables represent the same object in memory, or whether the values of one or more of their fields are equivalent. If you're intending to compare values, you must consider whether the objects are instances of value types (structs) or reference types (classes, delegates, arrays).

- To determine whether two class instances refer to the same location in memory (which means that they have the same *identity*), use the static [Object.Equals](#) method. ([System.Object](#) is the implicit base class for all value types and reference types, including user-defined structs and classes.)
- To determine whether the instance fields in two struct instances have the same values, use the [ValueType.Equals](#) method. Because all structs implicitly inherit from [System.ValueType](#), you call the method directly on your object as shown in the following example:

```
// Person is defined in the previous example.

//public struct Person
//{
//    public string Name;
//    public int Age;
//    public Person(string name, int age)
//    {
//        Name = name;
//        Age = age;
//    }
//}

Person p1 = new Person("Wallace", 75);
Person p2 = new Person("", 42);
p2.Name = "Wallace";
p2.Age = 75;

if (p2.Equals(p1))
    Console.WriteLine("p2 and p1 have the same values.");

// Output: p2 and p1 have the same values.
```

The [System.ValueType](#) implementation of `Equals` uses boxing and reflection in some cases. For information about how to provide an efficient equality algorithm that is specific to your type, see [How to define value equality for a type](#). Records are reference types that use value semantics for equality.

- To determine whether the values of the fields in two class instances are equal, you might be able to use the [Equals](#) method or the `== operator`. However, only use them if the class has overridden or overloaded them to provide a custom definition of what "equality" means for objects of that type. The class might also implement the [IEquatable<T>](#) interface or the [IEqualityComparer<T>](#) interface. Both interfaces provide methods that can be used to test value equality. When designing your own classes that override `Equals`, make sure to follow the guidelines stated in [How to define value equality for a type](#) and [Object.Equals\(Object\)](#).

Related Sections

For more information:

- [Classes](#)
- [Constructors](#)
- [Finalizers](#)
- [Events](#)
- [object](#)
- [Inheritance](#)
- [class](#)

- Structure types
- new Operator
- Common Type System

Inheritance - derive types to create more specialized behavior

12/28/2021 • 6 minutes to read • [Edit Online](#)

Inheritance, together with encapsulation and polymorphism, is one of the three primary characteristics of object-oriented programming. Inheritance enables you to create new classes that reuse, extend, and modify the behavior defined in other classes. The class whose members are inherited is called the *base class*, and the class that inherits those members is called the *derived class*. A derived class can have only one direct base class. However, inheritance is transitive. If `ClassC` is derived from `ClassB`, and `ClassB` is derived from `ClassA`, `ClassC` inherits the members declared in `ClassB` and `ClassA`.

NOTE

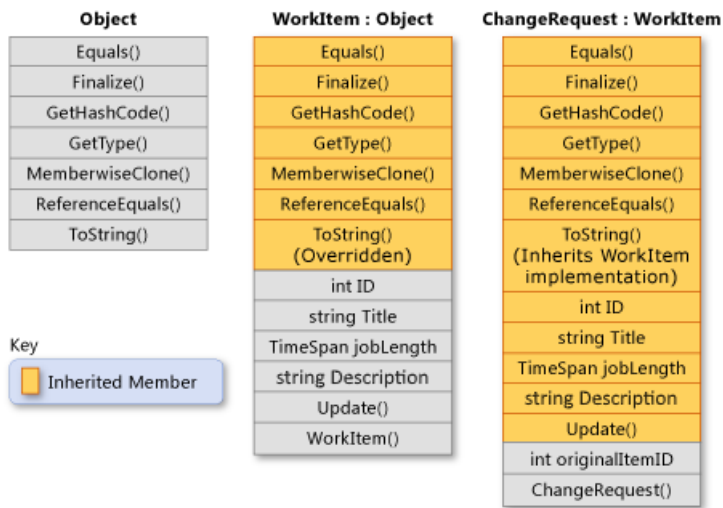
Structs do not support inheritance, but they can implement interfaces.

Conceptually, a derived class is a specialization of the base class. For example, if you have a base class `Animal`, you might have one derived class that is named `Mammal` and another derived class that is named `Reptile`. A `Mammal` is an `Animal`, and a `Reptile` is an `Animal`, but each derived class represents different specializations of the base class.

Interface declarations may define a default implementation for its members. These implementations are inherited by derived interfaces, and by classes that implement those interfaces. For more information on default interface methods, see the article on [interfaces](#).

When you define a class to derive from another class, the derived class implicitly gains all the members of the base class, except for its constructors and finalizers. The derived class reuses the code in the base class without having to reimplement it. You can add more members in the derived class. The derived class extends the functionality of the base class.

The following illustration shows a class `WorkItem` that represents an item of work in some business process. Like all classes, it derives from `System.Object` and inherits all its methods. `WorkItem` adds five members of its own. These members include a constructor, because constructors aren't inherited. Class `ChangeRequest` inherits from `WorkItem` and represents a particular kind of work item. `ChangeRequest` adds two more members to the members that it inherits from `WorkItem` and from `Object`. It must add its own constructor, and it also adds `originalItemID`. Property `originalItemID` enables the `ChangeRequest` instance to be associated with the original `WorkItem` to which the change request applies.



The following example shows how the class relationships demonstrated in the previous illustration are expressed in C#. The example also shows how `WorkItem` overrides the virtual method `Object.ToString`, and how the `ChangeRequest` class inherits the `WorkItem` implementation of the method. The first block defines the classes:

```
// WorkItem implicitly inherits from the Object class.
public class WorkItem
{
    // Static field currentID stores the job ID of the last WorkItem that
    // has been created.
    private static int currentID;

    //Properties.
    protected int ID { get; set; }
    protected string Title { get; set; }
    protected string Description { get; set; }
    protected TimeSpan jobLength { get; set; }

    // Default constructor. If a derived class does not invoke a base-
    // class constructor explicitly, the default constructor is called
    // implicitly.
    public WorkItem()
    {
        ID = 0;
        Title = "Default title";
        Description = "Default description.";
        jobLength = new TimeSpan();
    }

    // Instance constructor that has three parameters.
    public WorkItem(string title, string desc, TimeSpan joblen)
    {
        this.ID = GetNextID();
        this.Title = title;
        this.Description = desc;
        this.jobLength = joblen;
    }

    // Static constructor to initialize the static member, currentID. This
    // constructor is called one time, automatically, before any instance
    // of WorkItem or ChangeRequest is created, or currentID is referenced.
    static WorkItem() => currentID = 0;

    // currentID is a static field. It is incremented each time a new
    // instance of WorkItem is created.
    protected int GetNextID() => ++currentID;

    // Method Update enables you to update the title and job length of an
    // existing WorkItem object.
    public void Update(string title, TimeSpan joblen)
```

```

    {
        this.Title = title;
        this.jobLength = joblen;
    }

    // Virtual method override of the ToString method that is inherited
    // from System.Object.
    public override string ToString() =>
        $"{this.ID} - {this.Title}";
}

// ChangeRequest derives from WorkItem and adds a property (originalItemID)
// and two constructors.
public class ChangeRequest : WorkItem
{
    protected int originalItemID { get; set; }

    // Constructors. Because neither constructor calls a base-class
    // constructor explicitly, the default constructor in the base class
    // is called implicitly. The base class must contain a default
    // constructor.

    // Default constructor for the derived class.
    public ChangeRequest() { }

    // Instance constructor that has four parameters.
    public ChangeRequest(string title, string desc, TimeSpan jobLen,
        int originalID)
    {
        // The following properties and the GetNextID method are inherited
        // from WorkItem.
        this.ID = GetNextID();
        this.Title = title;
        this.Description = desc;
        this.jobLength = jobLen;

        // Property originalItemId is a member of ChangeRequest, but not
        // of WorkItem.
        this.originalItemID = originalID;
    }
}

```

This next block shows how to use the base and derived classes:

```
// Create an instance of WorkItem by using the constructor in the
// base class that takes three arguments.
WorkItem item = new WorkItem("Fix Bugs",
                             "Fix all bugs in my code branch",
                             new TimeSpan(3, 4, 0, 0));

// Create an instance of ChangeRequest by using the constructor in
// the derived class that takes four arguments.
ChangeRequest change = new ChangeRequest("Change Base Class Design",
                                         "Add members to the class",
                                         new TimeSpan(4, 0, 0, 0),
                                         1);

// Use the ToString method defined in WorkItem.
Console.WriteLine(item.ToString());

// Use the inherited Update method to change the title of the
// ChangeRequest object.
change.Update("Change the Design of the Base Class",
             new TimeSpan(4, 0, 0, 0));

// ChangeRequest inherits WorkItem's override of ToString.
Console.WriteLine(change.ToString());
/* Output:
    1 - Fix Bugs
    2 - Change the Design of the Base Class
*/
```

Abstract and virtual methods

When a base class declares a method as `virtual`, a derived class can `override` the method with its own implementation. If a base class declares a member as `abstract`, that method must be overridden in any non-abstract class that directly inherits from that class. If a derived class is itself abstract, it inherits abstract members without implementing them. Abstract and virtual members are the basis for polymorphism, which is the second primary characteristic of object-oriented programming. For more information, see [Polymorphism](#).

Abstract base classes

You can declare a class as `abstract` if you want to prevent direct instantiation by using the `new` operator. An abstract class can be used only if a new class is derived from it. An abstract class can contain one or more method signatures that themselves are declared as abstract. These signatures specify the parameters and return value but have no implementation (method body). An abstract class doesn't have to contain abstract members; however, if a class does contain an abstract member, the class itself must be declared as abstract. Derived classes that aren't abstract themselves must provide the implementation for any abstract methods from an abstract base class.

Interfaces

An *interface* is a reference type that defines a set of members. All classes and structs that implement that interface must implement that set of members. An interface may define a default implementation for any or all of these members. A class can implement multiple interfaces even though it can derive from only a single direct base class.

Interfaces are used to define specific capabilities for classes that don't necessarily have an "is a" relationship. For example, the `System.IEquatable<T>` interface can be implemented by any class or struct to determine whether two objects of the type are equivalent (however the type defines equivalence). `IEquatable<T>` doesn't imply the same kind of "is a" relationship that exists between a base class and a derived class (for example, a `Mammal` is an `Animal`). For more information, see [Interfaces](#).

Preventing further derivation

A class can prevent other classes from inheriting from it, or from any of its members, by declaring itself or the member as `sealed`.

Derived class hiding of base class members

A derived class can hide base class members by declaring members with the same name and signature. The `new` modifier can be used to explicitly indicate that the member isn't intended to be an override of the base member. The use of `new` isn't required, but a compiler warning will be generated if `new` isn't used. For more information, see [Versioning with the Override and New Keywords](#) and [Knowing When to Use Override and New Keywords](#).

Polymorphism

12/28/2021 • 7 minutes to read • [Edit Online](#)

Polymorphism is often referred to as the third pillar of object-oriented programming, after encapsulation and inheritance. Polymorphism is a Greek word that means "many-shaped" and it has two distinct aspects:

- At run time, objects of a derived class may be treated as objects of a base class in places such as method parameters and collections or arrays. When this polymorphism occurs, the object's declared type is no longer identical to its run-time type.
- Base classes may define and implement **virtual methods**, and derived classes can **override** them, which means they provide their own definition and implementation. At run-time, when client code calls the method, the CLR looks up the run-time type of the object, and invokes that override of the virtual method. In your source code you can call a method on a base class, and cause a derived class's version of the method to be executed.

Virtual methods enable you to work with groups of related objects in a uniform way. For example, suppose you have a drawing application that enables a user to create various kinds of shapes on a drawing surface. You don't know at compile time which specific types of shapes the user will create. However, the application has to keep track of all the various types of shapes that are created, and it has to update them in response to user mouse actions. You can use polymorphism to solve this problem in two basic steps:

1. Create a class hierarchy in which each specific shape class derives from a common base class.
2. Use a virtual method to invoke the appropriate method on any derived class through a single call to the base class method.

First, create a base class called `Shape`, and derived classes such as `Rectangle`, `Circle`, and `Triangle`. Give the `Shape` class a virtual method called `Draw`, and override it in each derived class to draw the particular shape that the class represents. Create a `List<Shape>` object and add a `Circle`, `Triangle`, and `Rectangle` to it.

```

public class Shape
{
    // A few example members
    public int X { get; private set; }
    public int Y { get; private set; }
    public int Height { get; set; }
    public int Width { get; set; }

    // Virtual method
    public virtual void Draw()
    {
        Console.WriteLine("Performing base class drawing tasks");
    }
}

public class Circle : Shape
{
    public override void Draw()
    {
        // Code to draw a circle...
        Console.WriteLine("Drawing a circle");
        base.Draw();
    }
}

public class Rectangle : Shape
{
    public override void Draw()
    {
        // Code to draw a rectangle...
        Console.WriteLine("Drawing a rectangle");
        base.Draw();
    }
}

public class Triangle : Shape
{
    public override void Draw()
    {
        // Code to draw a triangle...
        Console.WriteLine("Drawing a triangle");
        base.Draw();
    }
}

```

To update the drawing surface, use a [foreach](#) loop to iterate through the list and call the `Draw` method on each `Shape` object in the list. Even though each object in the list has a declared type of `Shape`, it's the run-time type (the overridden version of the method in each derived class) that will be invoked.


```
// Polymorphism at work #1: a Rectangle, Triangle and Circle
// can all be used wherever a Shape is expected. No cast is
// required because an implicit conversion exists from a derived
// class to its base class.
var shapes = new List<Shape>
{
    new Rectangle(),
    new Triangle(),
    new Circle()
};

// Polymorphism at work #2: the virtual method Draw is
// invoked on each of the derived classes, not the base class.
foreach (var shape in shapes)
{
    shape.Draw();
}
/* Output:
    Drawing a rectangle
    Performing base class drawing tasks
    Drawing a triangle
    Performing base class drawing tasks
    Drawing a circle
    Performing base class drawing tasks
*/
```

In C#, every type is polymorphic because all types, including user-defined types, inherit from [Object](#).

Polymorphism overview

Virtual members

When a derived class inherits from a base class, it gains all the methods, fields, properties, and events of the base class. The designer of the derived class has different choices for the behavior of virtual methods:

- The derived class may override virtual members in the base class, defining new behavior.
- The derived class may inherit the closest base class method without overriding it, preserving the existing behavior but enabling further derived classes to override the method.
- The derived class may define new non-virtual implementation of those members that hide the base class implementations.

A derived class can override a base class member only if the base class member is declared as [virtual](#) or [abstract](#). The derived member must use the [override](#) keyword to explicitly indicate that the method is intended to participate in virtual invocation. The following code provides an example:

```
public class BaseClass
{
    public virtual void DoWork() { }
    public virtual int WorkProperty
    {
        get { return 0; }
    }
}
public class DerivedClass : BaseClass
{
    public override void DoWork() { }
    public override int WorkProperty
    {
        get { return 0; }
    }
}
```

Fields can't be virtual; only methods, properties, events, and indexers can be virtual. When a derived class overrides a virtual member, that member is called even when an instance of that class is being accessed as an instance of the base class. The following code provides an example:

```
DerivedClass B = new DerivedClass();
B.DoWork(); // Calls the new method.

BaseClass A = B;
A.DoWork(); // Also calls the new method.
```

Virtual methods and properties enable derived classes to extend a base class without needing to use the base class implementation of a method. For more information, see [Versioning with the Override and New Keywords](#). An interface provides another way to define a method or set of methods whose implementation is left to derived classes.

Hide base class members with new members

If you want your derived class to have a member with the same name as a member in a base class, you can use the **new** keyword to hide the base class member. The **new** keyword is put before the return type of a class member that is being replaced. The following code provides an example:

```
public class BaseClass
{
    public void DoWork() { WorkField++; }
    public int WorkField;
    public int WorkProperty
    {
        get { return 0; }
    }
}

public class DerivedClass : BaseClass
{
    public new void DoWork() { WorkField++; }
    public new int WorkField;
    public new int WorkProperty
    {
        get { return 0; }
    }
}
```

Hidden base class members may be accessed from client code by casting the instance of the derived class to an instance of the base class. For example:

```
DerivedClass B = new DerivedClass();
B.DoWork(); // Calls the new method.

BaseClass A = (BaseClass)B;
A.DoWork(); // Calls the old method.
```

Prevent derived classes from overriding virtual members

Virtual members remain virtual, regardless of how many classes have been declared between the virtual member and the class that originally declared it. If class **A** declares a virtual member, and class **B** derives from **A**, and class **C** derives from **B**, class **C** inherits the virtual member, and may override it, regardless of whether class **B** declared an override for that member. The following code provides an example:

```
public class A
{
    public virtual void DoWork() { }
}
public class B : A
{
    public override void DoWork() { }
}
```

A derived class can stop virtual inheritance by declaring an override as [sealed](#). Stopping inheritance requires putting the `sealed` keyword before the `override` keyword in the class member declaration. The following code provides an example:

```
public class C : B
{
    public sealed override void DoWork() { }
}
```

In the previous example, the method `DoWork` is no longer virtual to any class derived from `C`. It's still virtual for instances of `C`, even if they're cast to type `B` or type `A`. Sealed methods can be replaced by derived classes by using the `new` keyword, as the following example shows:

```
public class D : C
{
    public new void DoWork() { }
}
```

In this case, if `DoWork` is called on `D` using a variable of type `D`, the new `DoWork` is called. If a variable of type `C`, `B`, or `A` is used to access an instance of `D`, a call to `DoWork` will follow the rules of virtual inheritance, routing those calls to the implementation of `DoWork` on class `C`.

Access base class virtual members from derived classes

A derived class that has replaced or overridden a method or property can still access the method or property on the base class using the `base` keyword. The following code provides an example:

```
public class Base
{
    public virtual void DoWork() { /*...*/ }
}
public class Derived : Base
{
    public override void DoWork()
    {
        //Perform Derived's work here
        //...
        // Call DoWork on base class
        base.DoWork();
    }
}
```

For more information, see [base](#).

NOTE

It is recommended that virtual members use `base` to call the base class implementation of that member in their own implementation. Letting the base class behavior occur enables the derived class to concentrate on implementing behavior specific to the derived class. If the base class implementation is not called, it is up to the derived class to make their behavior compatible with the behavior of the base class.

Pattern matching overview

12/28/2021 • 6 minutes to read • [Edit Online](#)

Pattern matching is a technique where you test an expression to determine if it has certain characteristics. C# pattern matching provides more concise syntax for testing expressions and taking action when an expression matches. The "`is`" expression supports pattern matching to test an expression and conditionally declare a new variable to the result of that expression. The "`switch`" expression enables you to perform actions based on the first matching pattern for an expression. These two expressions support a rich vocabulary of *patterns*.

This article provides an overview of scenarios where you can use pattern matching. These techniques can improve the readability and correctness of your code. For a full discussion of all the patterns you can apply, see the article on *patterns* in the language reference.

Null checks

One of the most common scenarios for pattern matching is to ensure values aren't `null`. You can test and convert a nullable value type to its underlying type while testing for `null` using the following example:

```
int? maybe = 12;

if (maybe is int number)
{
    Console.WriteLine($"The nullable int 'maybe' has the value {number}");
}
else
{
    Console.WriteLine("The nullable int 'maybe' doesn't hold a value");
}
```

The preceding code is a *declaration pattern* to test the type of the variable, and assign it to a new variable. The language rules make this technique safer than many others. The variable `number` is only accessible and assigned in the true portion of the `if` clause. If you try to access it elsewhere, either in the `else` clause, or after the `if` block, the compiler issues an error. Secondly, because you're not using the `==` operator, this pattern works when a type overloads the `==` operator. That makes it an ideal way to check null reference values, adding the `not` pattern:

```
string? message = "This is not the null string";

if (message is not null)
{
    Console.WriteLine(message);
}
```

The preceding example used a *constant pattern* to compare the variable to `null`. The `not` is a *logical pattern* that matches when the negated pattern doesn't match.

Type tests

Another common use for pattern matching is to test a variable to see if it matches a given type. For example, the following code tests if a variable is non-null and implements the `System.Collections.Generic.IList<T>` interface. If it does, it uses the `ICollection<T>.Count` property on that list to find the middle index. The declaration pattern

doesn't match a `null` value, regardless of the compile-time type of the variable. The code below guards against `null`, in addition to guarding against a type that doesn't implement `IList` .

```
public static T MidPoint<T>(IEnumerable<T> sequence)
{
    if (sequence is IList<T> list)
    {
        return list[list.Count / 2];
    }
    else if (sequence is null)
    {
        throw new ArgumentNullException(nameof(sequence), "Sequence can't be null.");
    }
    else
    {
        int halfLength = sequence.Count() / 2 - 1;
        if (halfLength < 0) halfLength = 0;
        return sequence.Skip(halfLength).First();
    }
}
```

The same tests can be applied in a `switch` expression to test a variable against multiple different types. You can use that information to create better algorithms based on the specific run-time type.

Compare discrete values

You can also test a variable to find a match on specific values. The following code shows one example where you test a value against all possible values declared in an enumeration:

```
public State PerformOperation(Operation command) =>
    command switch
    {
        Operation.SystemTest => RunDiagnostics(),
        Operation.Start => StartSystem(),
        Operation.Stop => StopSystem(),
        Operation.Reset => ResetToReady(),
        _ => throw new ArgumentException("Invalid enum value for command", nameof(command)),
    };
};
```

The previous example demonstrates a method dispatch based on the value of an enumeration. The final `_` case is a *discard pattern* that matches all values. It handles any error conditions where the value doesn't match one of the defined `enum` values. If you omit that switch arm, the compiler warns that you haven't handled all possible input values. At run time, the `switch` expression throws an exception if the object being examined doesn't match any of the switch arms. You could use numeric constants instead of a set of enum values. You can also use this similar technique for constant string values that represent the commands:

```
public State PerformOperation(string command) =>
    command switch
    {
        "SystemTest" => RunDiagnostics(),
        "Start" => StartSystem(),
        "Stop" => StopSystem(),
        "Reset" => ResetToReady(),
        _ => throw new ArgumentException("Invalid string value for command", nameof(command)),
    };
};
```

The preceding example shows the same algorithm, but uses string values instead of an enum. You would use this scenario if your application responds to text commands instead of a regular data format. In all these

examples, the *discard pattern* ensures that you handle every input. The compiler helps you by making sure every possible input value is handled.

Relational patterns

You can use *relational patterns* to test how a value compares to constants. For example, the following code returns the state of water based on the temperature in Fahrenheit:

```
string WaterState(int tempInFahrenheit) =>
    tempInFahrenheit switch
    {
        (> 32) and (< 212) => "liquid",
        < 32 => "solid",
        > 212 => "gas",
        32 => "solid/liquid transition",
        212 => "liquid / gas transition",
    };
```

The preceding code also demonstrates the conjunctive `and` *logical pattern* to check that both relational patterns match. You can also use a disjunctive `or` pattern to check that either pattern matches. The two relational patterns are surrounded by parentheses, which you can use around any pattern for clarity. The final two switch arms handle the cases for the melting point and the boiling point. Without those two arms, the compiler warns you that your logic doesn't cover every possible input.

The preceding code also demonstrates another important feature the compiler provides for pattern matching expressions: The compiler warns you if you don't handle every input value. The compiler also issues a warning if a switch arm is already handled by a previous switch arm. That gives you freedom to refactor and reorder switch expressions. Another way to write the same expression could be:

```
string WaterState2(int tempInFahrenheit) =>
    tempInFahrenheit switch
    {
        < 32 => "solid",
        32 => "solid/liquid transition",
        < 212 => "liquid",
        212 => "liquid / gas transition",
        _ => "gas",
    };
```

The key lesson in this, and any other refactoring or reordering is that the compiler validates that you've covered all inputs.

Multiple inputs

All the patterns you've seen so far have been checking one input. You can write patterns that examine multiple properties of an object. Consider the following `Order` record:

```
public record Order(int Items, decimal Cost);
```

The preceding positional record type declares two members at explicit positions. Appearing first is the `Items`, then the order's `Cost`. For more information, see [Records](#).

The following code examines the number of items and the value of an order to calculate a discounted price:

```
public decimal CalculateDiscount(Order order) =>
    order switch
    {
        (Items: > 10, Cost: > 1000.00m) => 0.10m,
        (Items: > 5, Cost: > 500.00m) => 0.05m,
        Order { Cost: > 250.00m } => 0.02m,
        null => throw new ArgumentNullException(nameof(order), "Can't calculate discount on null order"),
        var someObject => 0m,
    };

```

The first two arms examine two properties of the `Order`. The third examines only the cost. The next checks against `null`, and the final matches any other value. If the `Order` type defines a suitable `Deconstruct` method, you can omit the property names from the pattern and use deconstruction to examine properties:

```
public decimal CalculateDiscount(Order order) =>
    order switch
    {
        ( > 10, > 1000.00m) => 0.10m,
        ( > 5, > 500.00m) => 0.05m,
        Order { Cost: > 250.00m } => 0.02m,
        null => throw new ArgumentNullException(nameof(order), "Can't calculate discount on null order"),
        var someObject => 0m,
    };

```

The preceding code demonstrates the *positional pattern* where the properties are deconstructed for the expression.

This article provided a tour of the kinds of code you can write with pattern matching in C#. The following articles show more examples of using patterns in scenarios, and the full vocabulary of patterns available to use.

See also

- [Exploration: Use pattern matching to build your class behavior for better code](#)
- [Tutorial: Use pattern matching to build type-driven and data-driven algorithms](#)
- [Reference: Pattern matching](#)

Discards - C# Fundamentals

12/28/2021 • 8 minutes to read • [Edit Online](#)

Starting with C# 7.0, C# supports discards, which are placeholder variables that are intentionally unused in application code. Discards are equivalent to unassigned variables; they don't have a value. A discard communicates intent to the compiler and others that read your code: You intended to ignore the result of an expression. You may want to ignore the result of an expression, one or more members of a tuple expression, an `out` parameter to a method, or the target of a pattern matching expression.

Because there's only a single discard variable, that variable may not even be allocated storage. Discards can reduce memory allocations. Discards make the intent of your code clear. They enhance its readability and maintainability.

You indicate that a variable is a discard by assigning it the underscore (`_`) as its name. For example, the following method call returns a tuple in which the first and second values are discards. `area` is a previously declared variable set to the third component returned by `GetCityInformation`:

```
(_, _, area) = city.GetCityInformation(cityName);
```

Beginning with C# 9.0, you can use discards to specify unused input parameters of a lambda expression. For more information, see the [Input parameters of a lambda expression](#) section of the [Lambda expressions](#) article.

When `_` is a valid discard, attempting to retrieve its value or use it in an assignment operation generates compiler error CS0301, "The name '_' doesn't exist in the current context". This error is because `_` isn't assigned a value, and may not even be assigned a storage location. If it were an actual variable, you couldn't discard more than one value, as the previous example did.

Tuple and object deconstruction

Discards are useful in working with tuples when your application code uses some tuple elements but ignores others. For example, the following `QueryCityDataForYears` method returns a tuple with the name of a city, its area, a year, the city's population for that year, a second year, and the city's population for that second year. The example shows the change in population between those two years. Of the data available from the tuple, we're unconcerned with the city area, and we know the city name and the two dates at design-time. As a result, we're only interested in the two population values stored in the tuple, and can handle its remaining values as discards.

```

var (_, _, _, pop1, _, pop2) = QueryCityDataForYears("New York City", 1960, 2010);

Console.WriteLine($"Population change, 1960 to 2010: {pop2 - pop1:N0}");

static (string, double, int, int, int, int) QueryCityDataForYears(string name, int year1, int year2)
{
    int population1 = 0, population2 = 0;
    double area = 0;

    if (name == "New York City")
    {
        area = 468.48;
        if (year1 == 1960)
        {
            population1 = 7781984;
        }
        if (year2 == 2010)
        {
            population2 = 8175133;
        }
        return (name, area, year1, population1, year2, population2);
    }

    return ("", 0, 0, 0, 0, 0);
}

// The example displays the following output:
//      Population change, 1960 to 2010: 393,149

```

For more information on deconstructing tuples with discards, see [Deconstructing tuples and other types](#).

The `Deconstruct` method of a class, structure, or interface also allows you to retrieve and deconstruct a specific set of data from an object. You can use discards when you're interested in working with only a subset of the deconstructed values. The following example deconstructs a `Person` object into four strings (the first and last names, the city, and the state), but discards the last name and the state.

```

using System;

namespace Discards
{
    public class Person
    {
        public string FirstName { get; set; }
        public string MiddleName { get; set; }
        public string LastName { get; set; }
        public string City { get; set; }
        public string State { get; set; }

        public Person(string fname, string mname, string lname,
            string cityName, string stateName)
        {
            FirstName = fname;
            MiddleName = mname;
            LastName = lname;
            City = cityName;
            State = stateName;
        }

        // Return the first and last name.
        public void Deconstruct(out string fname, out string lname)
        {
            fname = FirstName;
            lname = LastName;
        }

        public void Deconstruct(out string fname, out string mname, out string lname)
        {
            fname = FirstName;
            mname = MiddleName;
            lname = LastName;
        }

        public void Deconstruct(out string fname, out string lname,
            out string city, out string state)
        {
            fname = FirstName;
            lname = LastName;
            city = City;
            state = State;
        }
    }
    class Example
    {
        public static void Main()
        {
            var p = new Person("John", "Quincy", "Adams", "Boston", "MA");

            // Deconstruct the person object.
            var (fname, _, city, _) = p;
            Console.WriteLine($"Hello {fname} of {city}!");
            // The example displays the following output:
            //      Hello John of Boston!
        }
    }
}

```

For more information on deconstructing user-defined types with discards, see [Deconstructing tuples and other types](#).

Pattern matching with `switch`

The *discard pattern* can be used in pattern matching with the [switch expression](#). Every expression, including `null`, always matches the discard pattern.

The following example defines a `ProvidesFormatInfo` method that uses a `switch` expression to determine whether an object provides an [IFormatProvider](#) implementation and tests whether the object is `null`. It also uses the discard pattern to handle non-null objects of any other type.

```
object[] objects = { CultureInfo.CurrentCulture,
                    CultureInfo.CurrentCulture.DateTimeFormat,
                    CultureInfo.CurrentCulture.NumberFormat,
                    new ArgumentException(), null };
foreach (var obj in objects)
    ProvidesFormatInfo(obj);

static void ProvidesFormatInfo(object obj) =>
    Console.WriteLine(obj switch
    {
        IFormatProvider fmt => $"{fmt.GetType()} object",
        null => "A null object reference: Its use could result in a NullReferenceException",
        _ => "Some object type without format information"
    });
// The example displays the following output:
// System.Globalization.CultureInfo object
// System.Globalization.DateTimeFormatInfo object
// System.Globalization.NumberFormatInfo object
// Some object type without format information
// A null object reference: Its use could result in a NullReferenceException
```

Calls to methods with `out` parameters

When calling the `Deconstruct` method to deconstruct a user-defined type (an instance of a class, structure, or interface), you can discard the values of individual `out` arguments. But you can also discard the value of `out` arguments when calling any method with an `out` parameter.

The following example calls the `DateTime.TryParse(String, out DateTime)` method to determine whether the string representation of a date is valid in the current culture. Because the example is concerned only with validating the date string and not with parsing it to extract the date, the `out` argument to the method is a discard.

```
string[] dateStrings = {"05/01/2018 14:57:32.8", "2018-05-01 14:57:32.8",
                        "2018-05-01T14:57:32.8375298-04:00", "5/01/2018",
                        "5/01/2018 14:57:32.80 -07:00",
                        "1 May 2018 2:57:32.8 PM", "16-05-2018 1:00:32 PM",
                        "Fri, 15 May 2018 20:10:57 GMT" };
foreach (string dateString in dateStrings)
{
    if (DateTime.TryParse(dateString, out _))
        Console.WriteLine($"'{dateString}': valid");
    else
        Console.WriteLine($"'{dateString}': invalid");
}
// The example displays output like the following:
// '05/01/2018 14:57:32.8': valid
// '2018-05-01 14:57:32.8': valid
// '2018-05-01T14:57:32.8375298-04:00': valid
// '5/01/2018': valid
// '5/01/2018 14:57:32.80 -07:00': valid
// '1 May 2018 2:57:32.8 PM': valid
// '16-05-2018 1:00:32 PM': invalid
// 'Fri, 15 May 2018 20:10:57 GMT': invalid
```

A standalone discard

You can use a standalone discard to indicate any variable that you choose to ignore. One typical use is to use an assignment to ensure that an argument isn't null. The following code uses a discard to force an assignment. The right side of the assignment uses the [null coalescing operator](#) to throw an [System.ArgumentNullException](#) when the argument is `null`. The code doesn't need the result of the assignment, so it's discarded. The expression forces a null check. The discard clarifies your intent: the result of the assignment isn't needed or used.

```
public static void Method(string arg)
{
    _ = arg ?? throw new ArgumentNullException(paramName: nameof(arg), message: "arg can't be null");

    // Do work with arg.
}
```

The following example uses a standalone discard to ignore the [Task](#) object returned by an asynchronous operation. Assigning the task has the effect of suppressing the exception that the operation throws as it is about to complete. It makes your intent clear: You want to discard the `Task`, and ignore any errors generated from that asynchronous operation.

```
private static async Task ExecuteAsyncMethods()
{
    Console.WriteLine("About to launch a task...");
    _ = Task.Run(() =>
    {
        var iterations = 0;
        for (int ctr = 0; ctr < int.MaxValue; ctr++)
            iterations++;
        Console.WriteLine("Completed looping operation...");
        throw new InvalidOperationException();
    });
    await Task.Delay(5000);
    Console.WriteLine("Exiting after 5 second delay");
}
// The example displays output like the following:
//     About to launch a task...
//     Completed looping operation...
//     Exiting after 5 second delay
```

Without assigning the task to a discard, the following code generates a compiler warning:

```
private static async Task ExecuteAsyncMethods()
{
    Console.WriteLine("About to launch a task...");
    // CS4014: Because this call is not awaited, execution of the current method continues before the call
    // is completed.
    // Consider applying the 'await' operator to the result of the call.
    Task.Run(() =>
    {
        var iterations = 0;
        for (int ctr = 0; ctr < int.MaxValue; ctr++)
            iterations++;
        Console.WriteLine("Completed looping operation...");
        throw new InvalidOperationException();
    });
    await Task.Delay(5000);
    Console.WriteLine("Exiting after 5 second delay");
}
```

NOTE

If you run either of the preceding two samples using a debugger, the debugger will stop the program when the exception is thrown. Without a debugger attached, the exception is silently ignored in both cases.

`_` is also a valid identifier. When used outside of a supported context, `_` is treated not as a discard but as a valid variable. If an identifier named `_` is already in scope, the use of `_` as a standalone discard can result in:

- Accidental modification of the value of the in-scope `_` variable by assigning it the value of the intended discard. For example:

```
private static void ShowValue(int _)
{
    byte[] arr = { 0, 0, 1, 2 };
    _ = BitConverter.ToInt32(arr, 0);
    Console.WriteLine(_);
}
// The example displays the following output:
//      33619968
```

- A compiler error for violating type safety. For example:

```
private static bool RoundTrips(int _)
{
    string value = _.ToString();
    int newValue = 0;
    _ = Int32.TryParse(value, out newValue);
    return _ == newValue;
}
// The example displays the following compiler error:
//      error CS0029: Cannot implicitly convert type 'bool' to 'int'
```

- Compiler error CS0136, "A local or parameter named '_' cannot be declared in this scope because that name is used in an enclosing local scope to define a local or parameter." For example:

```
public void DoSomething(int _)
{
    var _ = GetValue(); // Error: cannot declare local _ when one is already in scope
}
// The example displays the following compiler error:
// error CS0136:
//      A local or parameter named '_' cannot be declared in this scope
//      because that name is used in an enclosing local scope
//      to define a local or parameter
```

See also

- [Deconstructing tuples and other types](#)
- [is operator](#)
- [switch expression](#)

Deconstructing tuples and other types

12/28/2021 • 10 minutes to read • [Edit Online](#)

A tuple provides a lightweight way to retrieve multiple values from a method call. But once you retrieve the tuple, you have to handle its individual elements. Working on an element-by-element basis is cumbersome, as the following example shows. The `QueryCityData` method returns a three-tuple, and each of its elements is assigned to a variable in a separate operation.

```
public class Example
{
    public static void Main()
    {
        var result = QueryCityData("New York City");

        var city = result.Item1;
        var pop = result.Item2;
        var size = result.Item3;

        // Do something with the data.
    }

    private static (string, int, double) QueryCityData(string name)
    {
        if (name == "New York City")
            return (name, 8175133, 468.48);

        return ("", 0, 0);
    }
}
```

Retrieving multiple field and property values from an object can be equally cumbersome: you must assign a field or property value to a variable on a member-by-member basis.

In C# 7.0 and later, you can retrieve multiple elements from a tuple or retrieve multiple field, property, and computed values from an object in a single *deconstruct* operation. To deconstruct a tuple, you assign its elements to individual variables. When you deconstruct an object, you assign selected values to individual variables.

Tuples

C# features built-in support for deconstructing tuples, which lets you unpack all the items in a tuple in a single operation. The general syntax for deconstructing a tuple is similar to the syntax for defining one: you enclose the variables to which each element is to be assigned in parentheses in the left side of an assignment statement. For example, the following statement assigns the elements of a four-tuple to four separate variables:

```
var (name, address, city, zip) = contact.GetAddressInfo();
```

There are three ways to deconstruct a tuple:

- You can explicitly declare the type of each field inside parentheses. The following example uses this approach to deconstruct the three-tuple returned by the `QueryCityData` method.

```
public static void Main()
{
    (string city, int population, double area) = QueryCityData("New York City");

    // Do something with the data.
}
```

- You can use the `var` keyword so that C# infers the type of each variable. You place the `var` keyword outside of the parentheses. The following example uses type inference when deconstructing the three-tuple returned by the `QueryCityData` method.

```
public static void Main()
{
    var (city, population, area) = QueryCityData("New York City");

    // Do something with the data.
}
```

You can also use the `var` keyword individually with any or all of the variable declarations inside the parentheses.

```
public static void Main()
{
    (string city, var population, var area) = QueryCityData("New York City");

    // Do something with the data.
}
```

This is cumbersome and isn't recommended.

- Lastly, you may deconstruct the tuple into variables that have already been declared.

```
public static void Main()
{
    string city = "Raleigh";
    int population = 458880;
    double area = 144.8;

    (city, population, area) = QueryCityData("New York City");

    // Do something with the data.
}
```

- Beginning in C# 10, you can mix variable declaration and assignment in a deconstruction.

```
public static void Main()
{
    string city = "Raleigh";
    int population = 458880;

    (city, population, double area) = QueryCityData("New York City");

    // Do something with the data.
}
```

You can't specify a specific type outside the parentheses even if every field in the tuple has the same type. Doing so generates compiler error CS8136, "Deconstruction 'var (...)' form disallows a specific type for 'var'".

You must assign each element of the tuple to a variable. If you omit any elements, the compiler generates error CS8132, "Can't deconstruct a tuple of 'x' elements into 'y' variables."

Tuple elements with discards

Often when deconstructing a tuple, you're interested in the values of only some elements. Starting with C# 7.0, you can take advantage of C#'s support for *discards*, which are write-only variables whose values you've chosen to ignore. A discard is chosen by an underscore character ("_") in an assignment. You can discard as many values as you like; all are represented by the single discard, `_`.

The following example illustrates the use of tuples with discards. The `QueryCityDataForYears` method returns a six-tuple with the name of a city, its area, a year, the city's population for that year, a second year, and the city's population for that second year. The example shows the change in population between those two years. Of the data available from the tuple, we're unconcerned with the city area, and we know the city name and the two dates at design-time. As a result, we're only interested in the two population values stored in the tuple, and can handle its remaining values as discards.

```
using System;

public class ExampleDiscard
{
    public static void Main()
    {
        var (_, _, _, pop1, _, pop2) = QueryCityDataForYears("New York City", 1960, 2010);

        Console.WriteLine($"Population change, 1960 to 2010: {pop2 - pop1:N0}");
    }

    private static (string, double, int, int, int, int) QueryCityDataForYears(string name, int year1, int
year2)
    {
        int population1 = 0, population2 = 0;
        double area = 0;

        if (name == "New York City")
        {
            area = 468.48;
            if (year1 == 1960)
            {
                population1 = 7781984;
            }
            if (year2 == 2010)
            {
                population2 = 8175133;
            }
            return (name, area, year1, population1, year2, population2);
        }

        return ("", 0, 0, 0, 0, 0);
    }
}

// The example displays the following output:
//      Population change, 1960 to 2010: 393,149
```

User-defined types

C# doesn't offer built-in support for deconstructing non-tuple types other than the `record` and `DictionaryEntry` types. However, as the author of a class, a struct, or an interface, you can allow instances of the type to be deconstructed by implementing one or more `Deconstruct` methods. The method returns void, and each value to be deconstructed is indicated by an `out` parameter in the method signature. For example, the following

`Deconstruct` method of a `Person` class returns the first, middle, and last name:

```
public void Deconstruct(out string fname, out string mname, out string lname)
```

You can then deconstruct an instance of the `Person` class named `p` with an assignment like the following code:

```
var (fName, mName, lName) = p;
```

The following example overloads the `Deconstruct` method to return various combinations of properties of a `Person` object. Individual overloads return:

- A first and last name.
- A first, middle, and last name.
- A first name, a last name, a city name, and a state name.

```

using System;

public class Person
{
    public string FirstName { get; set; }
    public string MiddleName { get; set; }
    public string LastName { get; set; }
    public string City { get; set; }
    public string State { get; set; }

    public Person(string fname, string mname, string lname,
                  string cityName, string stateName)
    {
        FirstName = fname;
        MiddleName = mname;
        LastName = lname;
        City = cityName;
        State = stateName;
    }

    // Return the first and last name.
    public void Deconstruct(out string fname, out string lname)
    {
        fname = FirstName;
        lname = LastName;
    }

    public void Deconstruct(out string fname, out string mname, out string lname)
    {
        fname = FirstName;
        mname = MiddleName;
        lname = LastName;
    }

    public void Deconstruct(out string fname, out string lname,
                            out string city, out string state)
    {
        fname = FirstName;
        lname = LastName;
        city = City;
        state = State;
    }
}

public class ExampleClassDeconstruction
{
    public static void Main()
    {
        var p = new Person("John", "Quincy", "Adams", "Boston", "MA");

        // Deconstruct the person object.
        var (fName, lName, city, state) = p;
        Console.WriteLine($"Hello {fName} {lName} of {city}, {state}!");
    }
}

// The example displays the following output:
//   Hello John Adams of Boston, MA!

```

Multiple `Deconstruct` methods having the same number of parameters are ambiguous. You must be careful to define `Deconstruct` methods with different numbers of parameters, or "arity". `Deconstruct` methods with the same number of parameters cannot be distinguished during overload resolution.

User-defined type with discards

Just as you do with [tuples](#), you can use discards to ignore selected items returned by a `Deconstruct` method.

Each discard is defined by a variable named "_", and a single deconstruction operation can include multiple discards.

The following example deconstructs a `Person` object into four strings (the first and last names, the city, and the state) but discards the last name and the state.

```
// Deconstruct the person object.
var (fName, _, city, _) = p;
Console.WriteLine($"Hello {fName} of {city}!");
// The example displays the following output:
//      Hello John of Boston!
```

Extension methods for user-defined types

If you didn't author a class, struct, or interface, you can still deconstruct objects of that type by implementing one or more `Deconstruct` [extension methods](#) to return the values in which you're interested.

The following example defines two `Deconstruct` extension methods for the `System.Reflection.PropertyInfo` class. The first returns a set of values that indicate the characteristics of the property, including its type, whether it's static or instance, whether it's read-only, and whether it's indexed. The second indicates the property's accessibility. Because the accessibility of get and set accessors can differ, Boolean values indicate whether the property has separate get and set accessors and, if it does, whether they have the same accessibility. If there's only one accessor or both the get and the set accessor have the same accessibility, the `access` variable indicates the accessibility of the property as a whole. Otherwise, the accessibility of the get and set accessors are indicated by the `getAccess` and `setAccess` variables.

```
using System;
using System.Collections.Generic;
using System.Reflection;

public static class ReflectionExtensions
{
    public static void Deconstruct(this PropertyInfo p, out bool isStatic,
                                   out bool isReadOnly, out bool isIndexed,
                                   out Type propertyType)
    {
        var getter = p.GetMethod;

        // Is the property read-only?
        isReadOnly = ! p.CanWrite;

        // Is the property instance or static?
        isStatic = getter.IsStatic;

        // Is the property indexed?
        isIndexed = p.GetIndexParameters().Length > 0;

        // Get the property type.
        propertyType = p.PropertyType;
    }

    public static void Deconstruct(this PropertyInfo p, out bool hasGetAndSet,
                                   out bool sameAccess, out string access,
                                   out string getAccess, out string setAccess)
    {
        hasGetAndSet = sameAccess = false;
        string getAccessTemp = null;
        string setAccessTemp = null;

        MethodInfo getter = null;
        if (p.CanRead)
            getter = p.GetMethod;
```

```

        getter = p.GetMethod();

        MethodInfo setter = null;
        if (p.CanWrite)
            setter = p.SetMethod();

        if (setter != null && getter != null)
            hasGetAndSet = true;

        if (getter != null)
        {
            if (getter.IsPublic)
                getAccessTemp = "public";
            else if (getter.IsPrivate)
                getAccessTemp = "private";
            else if (getter.IsAssembly)
                getAccessTemp = "internal";
            else if (getter.IsFamily)
                getAccessTemp = "protected";
            else if (getter.IsFamilyOrAssembly)
                getAccessTemp = "protected internal";
        }

        if (setter != null)
        {
            if (setter.IsPublic)
                setAccessTemp = "public";
            else if (setter.IsPrivate)
                setAccessTemp = "private";
            else if (setter.IsAssembly)
                setAccessTemp = "internal";
            else if (setter.IsFamily)
                setAccessTemp = "protected";
            else if (setter.IsFamilyOrAssembly)
                setAccessTemp = "protected internal";
        }

        // Are the accessibility of the getter and setter the same?
        if (setAccessTemp == getAccessTemp)
        {
            sameAccess = true;
            access = getAccessTemp;
            getAccess = setAccess = String.Empty;
        }
        else
        {
            access = null;
            getAccess = getAccessTemp;
            setAccess = setAccessTemp;
        }
    }
}

public class ExampleExtension
{
    public static void Main()
    {
        Type dateType = typeof(DateTime);
        PropertyInfo prop = dateType.GetProperty("Now");
        var (isStatic, isRO, isIndexed, propType) = prop;
        Console.WriteLine($"The {dateType.FullName}.{prop.Name} property:");
        Console.WriteLine($"    PropertyType: {propType.Name}");
        Console.WriteLine($"    Static:      {isStatic}");
        Console.WriteLine($"    Read-only:   {isRO}");
        Console.WriteLine($"    Indexed:     {isIndexed}");

        Type listType = typeof(List<>);
        prop = listType.GetProperty("Item",

```

BindingFlags.Public | BindingFlags.NonPublic | BindingFlags.Instance |

```

BindingFlags.Static);
    var (hasGetAndSet, sameAccess, accessibility, getAccessibility, setAccessibility) = prop;
    Console.WriteLine($"Accessibility of the {listType.FullName}.{prop.Name} property: ");

    if (!hasGetAndSet | sameAccess)
    {
        Console.WriteLine(accessibility);
    }
    else
    {
        Console.WriteLine($"The get accessor: {getAccessibility}");
        Console.WriteLine($"The set accessor: {setAccessibility}");
    }
}
}

// The example displays the following output:
//      The System.DateTime.Now property:
//      PropertyType: DateTime
//      Static:      True
//      Read-only:   True
//      Indexed:     False
//
//      Accessibility of the System.Collections.Generic.List`1.Item property: public

```

Extension method for system types

Some system types provide the `Deconstruct` method as a convenience. For example, the `System.Collections.Generic.KeyValuePair<TKey,TValue>` type provides this functionality. When you're iterating over a `System.Collections.Generic.Dictionary<TKey,TValue>` each element is a `KeyValuePair<TKey, TValue>` and can be deconstructed. Consider the following example:

```

Dictionary<string, int> snapshotCommitMap = new(StringComparer.OrdinalIgnoreCase)
{
    ["https://github.com/dotnet/docs"] = 16_465,
    ["https://github.com/dotnet/runtime"] = 114_223,
    ["https://github.com/dotnet/installer"] = 22_436,
    ["https://github.com/dotnet/roslyn"] = 79_484,
    ["https://github.com/dotnet/aspnetcore"] = 48_386
};

foreach (var (repo, commitCount) in snapshotCommitMap)
{
    Console.WriteLine(
        $"The {repo} repository had {commitCount:N0} commits as of November 10th, 2021.");
}

```

You can add a `Deconstruct` method to system types that don't have one. Consider the following extension method:

```

public static class NullableExtensions
{
    public static void Deconstruct<T>(
        this T? nullable,
        out bool hasValue,
        out T value) where T : struct
    {
        hasValue = nullable.HasValue;
        value = nullable.GetValueOrDefault();
    }
}

```

This extension method allows all `Nullable<T>` types to be deconstructed into a tuple of

`(bool hasValue, T value)`. The following example shows code that uses this extension method:

```
DateTime? questionableDateTime = default;
var (hasValue, value) = questionableDateTime;
Console.WriteLine(
    $"{{ HasValue = {hasValue}, Value = {value} }}");

questionableDateTime = DateTime.Now;
(hasValue, value) = questionableDateTime;
Console.WriteLine(
    $"{{ HasValue = {hasValue}, Value = {value} }}");

// Example outputs:
// { HasValue = False, Value = 1/1/0001 12:00:00 AM }
// { HasValue = True, Value = 11/10/2021 6:11:45 PM }
```

record types

When you declare a [record](#) type by using two or more positional parameters, the compiler creates a `Deconstruct` method with an `out` parameter for each positional parameter in the `record` declaration. For more information, see [Positional syntax for property definition](#) and [Deconstructor behavior in derived records](#).

See also

- [Discards](#)
- [Tuple types](#)

Exceptions and Exception Handling

12/28/2021 • 2 minutes to read • [Edit Online](#)

The C# language's exception handling features help you deal with any unexpected or exceptional situations that occur when a program is running. Exception handling uses the `try`, `catch`, and `finally` keywords to try actions that may not succeed, to handle failures when you decide that it's reasonable to do so, and to clean up resources afterward. Exceptions can be generated by the common language runtime (CLR), by .NET or third-party libraries, or by application code. Exceptions are created by using the `throw` keyword.

In many cases, an exception may be thrown not by a method that your code has called directly, but by another method further down in the call stack. When an exception is thrown, the CLR will unwind the stack, looking for a method with a `catch` block for the specific exception type, and it will execute the first such `catch` block that it finds. If it finds no appropriate `catch` block anywhere in the call stack, it will terminate the process and display a message to the user.

In this example, a method tests for division by zero and catches the error. Without the exception handling, this program would terminate with a **DivideByZeroException was unhandled** error.

```
public class ExceptionTest
{
    static double SafeDivision(double x, double y)
    {
        if (y == 0)
            throw new DivideByZeroException();
        return x / y;
    }

    public static void Main()
    {
        // Input for test purposes. Change the values to see
        // exception handling behavior.
        double a = 98, b = 0;
        double result;

        try
        {
            result = SafeDivision(a, b);
            Console.WriteLine("{0} divided by {1} = {2}", a, b, result);
        }
        catch (DivideByZeroException)
        {
            Console.WriteLine("Attempted divide by zero.");
        }
    }
}
```

Exceptions Overview

Exceptions have the following properties:

- Exceptions are types that all ultimately derive from `System.Exception`.
- Use a `try` block around the statements that might throw exceptions.
- Once an exception occurs in the `try` block, the flow of control jumps to the first associated exception handler that is present anywhere in the call stack. In C#, the `catch` keyword is used to define an exception handler.

- If no exception handler for a given exception is present, the program stops executing with an error message.
- Don't catch an exception unless you can handle it and leave the application in a known state. If you catch `System.Exception`, rethrow it using the `throw` keyword at the end of the `catch` block.
- If a `catch` block defines an exception variable, you can use it to obtain more information about the type of exception that occurred.
- Exceptions can be explicitly generated by a program by using the `throw` keyword.
- Exception objects contain detailed information about the error, such as the state of the call stack and a text description of the error.
- Code in a `finally` block is executed regardless of if an exception is thrown. Use a `finally` block to release resources, for example to close any streams or files that were opened in the `try` block.
- Managed exceptions in .NET are implemented on top of the Win32 structured exception handling mechanism. For more information, see [Structured Exception Handling \(C/C++\)](#) and [A Crash Course on the Depths of Win32 Structured Exception Handling](#).

C# Language Specification

For more information, see [Exceptions](#) in the [C# Language Specification](#). The language specification is the definitive source for C# syntax and usage.

See also

- [SystemException](#)
- [C# Keywords](#)
- [throw](#)
- [try-catch](#)
- [try-finally](#)
- [try-catch-finally](#)
- [Exceptions](#)

Use exceptions

12/28/2021 • 3 minutes to read • [Edit Online](#)

In C#, errors in the program at run time are propagated through the program by using a mechanism called exceptions. Exceptions are thrown by code that encounters an error and caught by code that can correct the error. Exceptions can be thrown by the .NET runtime or by code in a program. Once an exception is thrown, it propagates up the call stack until a `catch` statement for the exception is found. Uncaught exceptions are handled by a generic exception handler provided by the system that displays a dialog box.

Exceptions are represented by classes derived from `Exception`. This class identifies the type of exception and contains properties that have details about the exception. Throwing an exception involves creating an instance of an exception-derived class, optionally configuring properties of the exception, and then throwing the object by using the `throw` keyword. For example:

```
class CustomException : Exception
{
    public CustomException(string message)
    {
    }
}
private static void TestThrow()
{
    throw new CustomException("Custom exception in TestThrow()");
}
```

After an exception is thrown, the runtime checks the current statement to see whether it is within a `try` block. If it is, any `catch` blocks associated with the `try` block are checked to see whether they can catch the exception. `catch` blocks typically specify exception types; if the type of the `catch` block is the same type as the exception, or a base class of the exception, the `catch` block can handle the method. For example:

```
try
{
    TestThrow();
}
catch (CustomException ex)
{
    System.Console.WriteLine(ex.ToString());
}
```

If the statement that throws an exception isn't within a `try` block or if the `try` block that encloses it has no matching `catch` block, the runtime checks the calling method for a `try` statement and `catch` blocks. The runtime continues up the calling stack, searching for a compatible `catch` block. After the `catch` block is found and executed, control is passed to the next statement after that `catch` block.

A `try` statement can contain more than one `catch` block. The first `catch` statement that can handle the exception is executed; any following `catch` statements, even if they're compatible, are ignored. Order catch blocks from most specific (or most-derived) to least specific. For example:

```

using System;
using System.IO;

namespace Exceptions
{
    public class CatchOrder
    {
        public static void Main()
        {
            try
            {
                using (var sw = new StreamWriter("./test.txt"))
                {
                    sw.WriteLine("Hello");
                }
            }
            // Put the more specific exceptions first.
            catch (DirectoryNotFoundException ex)
            {
                Console.WriteLine(ex);
            }
            catch (FileNotFoundException ex)
            {
                Console.WriteLine(ex);
            }
            // Put the least specific exception last.
            catch (IOException ex)
            {
                Console.WriteLine(ex);
            }
            Console.WriteLine("Done");
        }
    }
}

```

Before the `catch` block is executed, the runtime checks for `finally` blocks. `Finally` blocks enable the programmer to clean up any ambiguous state that could be left over from an aborted `try` block, or to release any external resources (such as graphics handles, database connections, or file streams) without waiting for the garbage collector in the runtime to finalize the objects. For example:

```

static void TestFinally()
{
    FileStream? file = null;
    //Change the path to something that works on your machine.
    FileInfo fileInfo = new System.IO.FileInfo("./file.txt");

    try
    {
        file = fileInfo.OpenWrite();
        file.WriteByte(0xF);
    }
    finally
    {
        // Closing the file allows you to reopen it immediately - otherwise IOException is thrown.
        file?.Close();
    }

    try
    {
        file = fileInfo.OpenWrite();
        Console.WriteLine("OpenWrite() succeeded");
    }
    catch (IOException)
    {
        Console.WriteLine("OpenWrite() failed");
    }
}

```

If `WriteByte()` threw an exception, the code in the second `try` block that tries to reopen the file would fail if `file.Close()` isn't called, and the file would remain locked. Because `finally` blocks are executed even if an exception is thrown, the `finally` block in the previous example allows for the file to be closed correctly and helps avoid an error.

If no compatible `catch` block is found on the call stack after an exception is thrown, one of three things occurs:

- If the exception is within a finalizer, the finalizer is aborted and the base finalizer, if any, is called.
- If the call stack contains a static constructor, or a static field initializer, a [TypeInitializationException](#) is thrown, with the original exception assigned to the [InnerException](#) property of the new exception.
- If the start of the thread is reached, the thread is terminated.

Exception Handling (C# Programming Guide)

12/28/2021 • 4 minutes to read • [Edit Online](#)

A `try` block is used by C# programmers to partition code that might be affected by an exception. Associated `catch` blocks are used to handle any resulting exceptions. A `finally` block contains code that is run whether or not an exception is thrown in the `try` block, such as releasing resources that are allocated in the `try` block. A `try` block requires one or more associated `catch` blocks, or a `finally` block, or both.

The following examples show a `try-catch` statement, a `try-finally` statement, and a `try-catch-finally` statement.

```
try
{
    // Code to try goes here.
}
catch (SomeSpecificException ex)
{
    // Code to handle the exception goes here.
    // Only catch exceptions that you know how to handle.
    // Never catch base class System.Exception without
    // rethrowing it at the end of the catch block.
}
```

```
try
{
    // Code to try goes here.
}
finally
{
    // Code to execute after the try block goes here.
}
```

```
try
{
    // Code to try goes here.
}
catch (SomeSpecificException ex)
{
    // Code to handle the exception goes here.
}
finally
{
    // Code to execute after the try (and possibly catch) blocks
    // goes here.
}
```

A `try` block without a `catch` or `finally` block causes a compiler error.

Catch Blocks

A `catch` block can specify the type of exception to catch. The type specification is called an *exception filter*. The exception type should be derived from `Exception`. In general, don't specify `Exception` as the exception filter unless either you know how to handle all exceptions that might be thrown in the `try` block, or you've included a `throw` statement at the end of your `catch` block.

Multiple `catch` blocks with different exception classes can be chained together. The `catch` blocks are evaluated from top to bottom in your code, but only one `catch` block is executed for each exception that is thrown. The first `catch` block that specifies the exact type or a base class of the thrown exception is executed. If no `catch` block specifies a matching exception class, a `catch` block that doesn't have any type is selected, if one is present in the statement. It's important to position `catch` blocks with the most specific (that is, the most derived) exception classes first.

Catch exceptions when the following conditions are true:

- You have a good understanding of why the exception might be thrown, and you can implement a specific recovery, such as prompting the user to enter a new file name when you catch a `FileNotFoundException` object.
- You can create and throw a new, more specific exception.

```
int GetInt(int[] array, int index)
{
    try
    {
        return array[index];
    }
    catch (IndexOutOfRangeException e)
    {
        throw new ArgumentOutOfRangeException(
            "Parameter index is out of range.", e);
    }
}
```

- You want to partially handle an exception before passing it on for more handling. In the following example, a `catch` block is used to add an entry to an error log before rethrowing the exception.

```
try
{
    // Try to access a resource.
}
catch (UnauthorizedAccessException e)
{
    // Call a custom error logging procedure.
    LogError(e);
    // Re-throw the error.
    throw;
}
```

You can also specify *exception filters* to add a boolean expression to a catch clause. Exception filters indicate that a specific catch clause matches only when that condition is true. In the following example, both catch clauses use the same exception class, but an extra condition is checked to create a different error message:

```
int GetInt(int[] array, int index)
{
    try
    {
        return array[index];
    }
    catch (IndexOutOfRangeException e) when (index < 0)
    {
        throw new ArgumentOutOfRangeException(
            "Parameter index cannot be negative.", e);
    }
    catch (IndexOutOfRangeException e)
    {
        throw new ArgumentOutOfRangeException(
            "Parameter index cannot be greater than the array size.", e);
    }
}
```

An exception filter that always returns `false` can be used to examine all exceptions but not process them. A typical use is to log exceptions:

```
public static void Main()
{
    try
    {
        string? s = null;
        Console.WriteLine(s.Length);
    }
    catch (Exception e) when (LogException(e))
    {
    }
    Console.WriteLine("Exception must have been handled");
}

private static bool LogException(Exception e)
{
    Console.WriteLine($"{e}\tIn the log routine. Caught {e.GetType()}");
    Console.WriteLine($"{e}\tMessage: {e.Message}");
    return false;
}
```

The `LogException` method always returns `false`, no `catch` clause using this exception filter matches. The catch clause can be general, using [System.Exception](#), and later clauses can process more specific exception classes.

Finally Blocks

A `finally` block enables you to clean up actions that are performed in a `try` block. If present, the `finally` block executes last, after the `try` block and any matched `catch` block. A `finally` block always runs, whether an exception is thrown or a `catch` block matching the exception type is found.

The `finally` block can be used to release resources such as file streams, database connections, and graphics handles without waiting for the garbage collector in the runtime to finalize the objects. For more information See the [using Statement](#).

In the following example, the `finally` block is used to close a file that is opened in the `try` block. Notice that the state of the file handle is checked before the file is closed. If the `try` block can't open the file, the file handle still has the value `null` and the `finally` block doesn't try to close it. Instead, if the file is opened successfully in the `try` block, the `finally` block closes the open file.

```
FileStream? file = null;
FileInfo fileinfo = new System.IO.FileInfo("./file.txt");
try
{
    file = fileinfo.OpenWrite();
    file.WriteByte(0xF);
}
finally
{
    // Check for null because OpenWrite might have failed.
    file?.Close();
}
```

C# Language Specification

For more information, see [Exceptions](#) and [The try statement](#) in the [C# Language Specification](#). The language specification is the definitive source for C# syntax and usage.

See also

- [C# Reference](#)
- [try-catch](#)
- [try-finally](#)
- [try-catch-finally](#)
- [using Statement](#)

Creating and Throwing Exceptions

12/28/2021 • 3 minutes to read • [Edit Online](#)

Exceptions are used to indicate that an error has occurred while running the program. Exception objects that describe an error are created and then *thrown* with the `throw` keyword. The runtime then searches for the most compatible exception handler.

Programmers should throw exceptions when one or more of the following conditions are true:

- The method can't complete its defined functionality. For example, if a parameter to a method has an invalid value:

```
static void CopyObject(SampleClass original)
{
    _ = original ?? throw new ArgumentException("Parameter cannot be null", nameof(original));
}
```

- An inappropriate call to an object is made, based on the object state. One example might be trying to write to a read-only file. In cases where an object state doesn't allow an operation, throw an instance of `InvalidOperationException` or an object based on a derivation of this class. The following code is an example of a method that throws an `InvalidOperationException` object:

```
public class ProgramLog
{
    FileStream logFile = null!;
    public void OpenLog(FileInfo fileName, FileMode mode) { }

    public void WriteLog()
    {
        if (!logFile.CanWrite)
        {
            throw new InvalidOperationException("Logfile cannot be read-only");
        }
        // Else write data to the log and return.
    }
}
```

- When an argument to a method causes an exception. In this case, the original exception should be caught and an `ArgumentException` instance should be created. The original exception should be passed to the constructor of the `ArgumentException` as the `InnerException` parameter:

```
static int GetValueFromArray(int[] array, int index)
{
    try
    {
        return array[index];
    }
    catch (IndexOutOfRangeException ex)
    {
        throw new ArgumentException("Index is out of range", nameof(index), ex);
    }
}
```

Exceptions contain a property named `StackTrace`. This string contains the name of the methods on the current

call stack, together with the file name and line number where the exception was thrown for each method. A [StackTrace](#) object is created automatically by the common language runtime (CLR) from the point of the `throw` statement, so that exceptions must be thrown from the point where the stack trace should begin.

All exceptions contain a property named [Message](#). This string should be set to explain the reason for the exception. Information that is sensitive to security shouldn't be put in the message text. In addition to [Message](#), [ArgumentException](#) contains a property named [ParamName](#) that should be set to the name of the argument that caused the exception to be thrown. In a property setter, [ParamName](#) should be set to `value`.

Public and protected methods throw exceptions whenever they can't complete their intended functions. The exception class thrown is the most specific exception available that fits the error conditions. These exceptions should be documented as part of the class functionality, and derived classes or updates to the original class should retain the same behavior for backward compatibility.

Things to Avoid When Throwing Exceptions

The following list identifies practices to avoid when throwing exceptions:

- Don't use exceptions to change the flow of a program as part of ordinary execution. Use exceptions to report and handle error conditions.
- Exceptions shouldn't be returned as a return value or parameter instead of being thrown.
- Don't throw [System.Exception](#), [System.SystemException](#), [System.NullReferenceException](#), or [System.IndexOutOfRangeException](#) intentionally from your own source code.
- Don't create exceptions that can be thrown in debug mode but not release mode. To identify run-time errors during the development phase, use `Debug.Assert` instead.

Defining Exception Classes

Programs can throw a predefined exception class in the [System](#) namespace (except where previously noted), or create their own exception classes by deriving from [Exception](#). The derived classes should define at least four constructors: one parameterless constructor, one that sets the message property, and one that sets both the [Message](#) and [InnerException](#) properties. The fourth constructor is used to serialize the exception. New exception classes should be serializable. For example:

```
[Serializable]
public class InvalidDepartmentException : Exception
{
    public InvalidDepartmentException() : base() { }
    public InvalidDepartmentException(string message) : base(message) { }
    public InvalidDepartmentException(string message, Exception inner) : base(message, inner) { }

    // A constructor is needed for serialization when an
    // exception propagates from a remoting server to the client.
    protected InvalidDepartmentException(System.Runtime.Serialization.SerializationInfo info,
        System.Runtime.Serialization.StreamingContext context) : base(info, context) { }
}
```

Add new properties to the exception class when the data they provide is useful to resolving the exception. If new properties are added to the derived exception class, `ToString()` should be overridden to return the added information.

C# Language Specification

For more information, see [Exceptions](#) and [The throw statement](#) in the [C# Language Specification](#). The language specification is the definitive source for C# syntax and usage.

See also

- [Exception Hierarchy](#)

Compiler-generated exceptions

12/28/2021 • 2 minutes to read • [Edit Online](#)

Some exceptions are thrown automatically by the .NET runtime when basic operations fail. These exceptions and their error conditions are listed in the following table.

EXCEPTION	DESCRIPTION
ArithmeticException	A base class for exceptions that occur during arithmetic operations, such as DivideByZeroException and OverflowException .
ArrayTypeMismatchException	Thrown when an array can't store a given element because the actual type of the element is incompatible with the actual type of the array.
DivideByZeroException	Thrown when an attempt is made to divide an integral value by zero.
IndexOutOfRangeException	Thrown when an attempt is made to index an array when the index is less than zero or outside the bounds of the array.
InvalidCastException	Thrown when an explicit conversion from a base type to an interface or to a derived type fails at run time.
NullReferenceException	Thrown when an attempt is made to reference an object whose value is null .
OutOfMemoryException	Thrown when an attempt to allocate memory using the new operator fails. This exception indicates that the memory available to the common language runtime has been exhausted.
OverflowException	Thrown when an arithmetic operation in a <code>checked</code> context overflows.
StackOverflowException	Thrown when the execution stack is exhausted by having too many pending method calls; usually indicates a very deep or infinite recursion.
TypeInitializationException	Thrown when a static constructor throws an exception and no compatible <code>catch</code> clause exists to catch it.

See also

- [try-catch](#)
- [try-finally](#)
- [try-catch-finally](#)

Identifier names

12/28/2021 • 2 minutes to read • [Edit Online](#)

An **identifier** is the name you assign to a type (class, interface, struct, delegate, or enum), member, variable, or namespace. Valid identifiers must follow these rules:

- Identifiers must start with a letter, or `_`.
- Identifiers may contain Unicode letter characters, decimal digit characters, Unicode connecting characters, Unicode combining characters, or Unicode formatting characters. For more information on Unicode categories, see the [Unicode Category Database](#). You can declare identifiers that match C# keywords by using the `@` prefix on the identifier. The `@` is not part of the identifier name. For example, `@if` declares an identifier named `if`. These [verbatim identifiers](#) are primarily for interoperability with identifiers declared in other languages.

For a complete definition of valid identifiers, see the [Identifiers topic in the C# Language Specification](#).

Naming conventions

In addition to the rules, there are many identifier [naming conventions](#) used throughout the .NET APIs. By convention, C# programs use `PascalCase` for type names, namespaces, and all public members. In addition, the following conventions are common:

- Interface names start with a capital `I`.
- Attribute types end with the word `Attribute`.
- Enum types use a singular noun for non-flags, and a plural noun for flags.
- Identifiers shouldn't contain two consecutive `_` characters. Those names are reserved for compiler-generated identifiers.

C# Language Specification

For more information, see the [C# Language Specification](#). The language specification is the definitive source for C# syntax and usage.

See also

- [C# Programming Guide](#)
- [C# Reference](#)
- [Classes](#)
- [Structure types](#)
- [Namespaces](#)
- [Interfaces](#)
- [Delegates](#)

C# Coding Conventions

12/28/2021 • 11 minutes to read • [Edit Online](#)

Coding conventions serve the following purposes:

- They create a consistent look to the code, so that readers can focus on content, not layout.
- They enable readers to understand the code more quickly by making assumptions based on previous experience.
- They facilitate copying, changing, and maintaining the code.
- They demonstrate C# best practices.

IMPORTANT

The guidelines in this article are used by Microsoft to develop samples and documentation. They were adopted from the [.NET Runtime, C# Coding Style](#) guidelines. You can use them, or adapt them to your needs. The primary objectives are consistency and readability within your project, team, organization, or company source code.

Naming conventions

There are several naming conventions to consider when writing C# code.

In the following examples, any of the guidance pertaining to elements marked `public` is also applicable when working with `protected` and `protected internal` elements, all of which are intended to be visible to external callers.

Pascal case

Use pascal casing ("PascalCasing") when naming a `class`, `record`, or `struct`.

```
public class DataService
{
}
```

```
public record PhysicalAddress(
    string Street,
    string City,
    string StateOrProvince,
    string ZipCode);
```

```
public struct ValueCoordinate
{
}
```

When naming an `interface`, use pascal casing in addition to prefixing the name with an `I`. This clearly indicates to consumers that it's an `interface`.

```
public interface IWorkerQueue
{
}
```

When naming `public` members of types, such as fields, properties, events, methods, and local functions, use pascal casing.

```
public class ExampleEvents
{
    // A public field, these should be used sparingly
    public bool IsValid;

    // An init-only property
    public IWorkerQueue WorkerQueue { get; init; }

    // An event
    public event Action EventProcessing;

    // Method
    public void StartEventProcessing()
    {
        // Local function
        static int CountQueueItems() => WorkerQueue.Count;
        // ...
    }
}
```

When writing positional records, use pascal casing for parameters as they're the public properties of the record.

```
public record PhysicalAddress(
    string Street,
    string City,
    string StateOrProvince,
    string ZipCode);
```

For more information on positional records, see [Positional syntax for property definition](#).

Camel case

Use camel casing ("camelCasing") when naming `private` or `internal` fields, and prefix them with `_`.

```
public class DataService
{
    private IWorkerQueue _workerQueue;
}
```

TIP

When editing C# code that follows these naming conventions in an IDE that supports statement completion, typing `_` will show all of the object-scoped members.

When working with `static` fields that are `private` or `internal`, use the `s_` prefix and for thread static use `t_`.

```
public class DataService
{
    private static IWorkerQueue s_workerQueue;

    [ThreadStatic]
    private static TimeSpan t_timeSpan;
}
```

When writing method parameters, use camel casing.

```
public T SomeMethod<T>(int someNumber, bool isValid)
{
}
```

For more information on C# naming conventions, see [C# Coding Style](#).

Additional naming conventions

- Examples that don't include [using directives](#), use namespace qualifications. If you know that a namespace is imported by default in a project, you don't have to fully qualify the names from that namespace. Qualified names can be broken after a dot (.) if they are too long for a single line, as shown in the following example.

```
var currentPerformanceCounterCategory = new System.Diagnostics.
    PerformanceCounterCategory();
```

- You don't have to change the names of objects that were created by using the Visual Studio designer tools to make them fit other guidelines.

Layout conventions

Good layout uses formatting to emphasize the structure of your code and to make the code easier to read. Microsoft examples and samples conform to the following conventions:

- Use the default Code Editor settings (smart indenting, four-character indents, tabs saved as spaces). For more information, see [Options, Text Editor, C#, Formatting](#).
- Write only one statement per line.
- Write only one declaration per line.
- If continuation lines are not indented automatically, indent them one tab stop (four spaces).
- Add at least one blank line between method definitions and property definitions.
- Use parentheses to make clauses in an expression apparent, as shown in the following code.

```
if ((val1 > val2) && (val1 > val3))
{
    // Take appropriate action.
}
```

Commenting conventions

- Place the comment on a separate line, not at the end of a line of code.
- Begin comment text with an uppercase letter.
- End comment text with a period.
- Insert one space between the comment delimiter (//) and the comment text, as shown in the following example.


```
// The following declaration creates a query. It does not run
// the query.
```

- Don't create formatted blocks of asterisks around comments.
- Ensure all public members have the necessary XML comments providing appropriate descriptions about their behavior.

Language guidelines

The following sections describe practices that the C# team follows to prepare code examples and samples.

String data type

- Use **string interpolation** to concatenate short strings, as shown in the following code.

```
string displayName = $"{nameList[n].LastName}, {nameList[n].FirstName}";
```

- To append strings in loops, especially when you're working with large amounts of text, use a [StringBuilder](#) object.

```
var phrase = "lalalalalalalalalalalalalalalalalalalalalalalalalalalal";  
var manyPhrases = new StringBuilder();  
for (var i = 0; i < 10000; i++)  
{  
    manyPhrases.Append(phrase);  
}  
  
//Console.WriteLine("tra" + manyPhrases);
```

Implicitly typed local variables

- Use **implicit typing** for local variables when the type of the variable is obvious from the right side of the assignment, or when the precise type is not important.

```
var var1 = "This is clearly a string.";
var var2 = 27;
```

- Don't use `var` when the type is not apparent from the right side of the assignment. Don't assume the type is clear from a method name. A variable type is considered clear if it's a `new` operator or an explicit cast.

```
int var3 = Convert.ToInt32(Console.ReadLine());
int var4 = ExampleClass.ResultSoFar();
```

- Don't rely on the variable name to specify the type of the variable. It might not be correct. In the following example, the variable name `inputInt` is misleading. It's a string.

```
var inputInt = Console.ReadLine();  
Console.WriteLine(inputInt);
```

- Avoid the use of `var` in place of `dynamic`. Use `dynamic` when you want run-time type inference. For more information, see [Using type dynamic \(C# Programming Guide\)](#).
- Use implicit typing to determine the type of the loop variable in `for` loops.

The following example uses implicit typing in a `for` statement.

```
var phrase = "lalalalalalalalalalalalalalalalalalalalalalalalala";  
var manyPhrases = new StringBuilder();  
for (var i = 0; i < 10000; i++)  
{  
    manyPhrases.Append(phrase);  
}  
  
//Console.WriteLine("tra" + manyPhrases);
```

- Don't use implicit typing to determine the type of the loop variable in `foreach` loops.

The following example uses explicit typing in a `foreach` statement.

```
foreach (char ch in laugh)
{
    if (ch == 'h')
        Console.Write("H");
    else
        Console.Write(ch);
}
Console.WriteLine();
```

NOTE

Be careful not to accidentally change a type of an element of the iterable collection. For example, it is easy to switch from `System.Linq.IQueryable` to `System.Collections.IEnumerable` in a `foreach` statement, which changes the execution of a query.

NOTE

Be careful not to accidentally change a type of an element of the iterable collection. For example, it is easy to switch from `System.Linq.IQueryable` to `System.Collections.IEnumerable` in a `foreach` statement, which changes the execution of a query.

Unsigned data types

In general, use `int` rather than unsigned types. The use of `int` is common throughout C#, and it is easier to interact with other libraries when you use `int`.

Arrays

Use the concise syntax when you initialize arrays on the declaration line. In the following example, note that you can't use `var` instead of `string[]`.

```
string[] vowels1 = { "a", "e", "i", "o", "u" };
```

If you use explicit instantiation, you can use `var`.

```
var vowels2 = new string[] { "a", "e", "i", "o", "u" };
```

If you specify an array size, you have to initialize the elements one at a time.

```
var vowels3 = new string[5];
vowels3[0] = "a";
vowels3[1] = "e";
// And so on.
```

Delegates

Use `Func<>` and `Action<>` instead of defining delegate types. In a class, define the delegate method.

```
public static Action<string> ActionExample1 = x => Console.WriteLine($"x is: {x}");

public static Action<string, string> ActionExample2 = (x, y) =>
    Console.WriteLine($"x is: {x}, y is {y}");

public static Func<string, int> FuncExample1 = x => Convert.ToInt32(x);

public static Func<int, int, int> FuncExample2 = (x, y) => x + y;
```

Call the method using the signature defined by the `Func<>` or `Action<>` delegate.

```
ActionExample1("string for x");

ActionExample2("string for x", "string for y");

Console.WriteLine($"The value is {FuncExample1("1")}");

Console.WriteLine($"The sum is {FuncExample2(1, 2)}");
```

If you create instances of a delegate type, use the concise syntax. In a class, define the delegate type and a method that has a matching signature.

```
public delegate void Del(string message);

public static void DelMethod(string str)
{
    Console.WriteLine("DelMethod argument: {0}", str);
}
```

Create an instance of the delegate type and call it. The following declaration shows the condensed syntax.

```
Del exampleDel2 = DelMethod;
exampleDel2("Hey");
```

The following declaration uses the full syntax.

```
Del exampleDel1 = new Del(DelMethod);
exampleDel1("Hey");
```

`try` - `catch` and `using` statements in exception handling

- Use a `try-catch` statement for most exception handling.

```
static string GetValueFromArray(string[] array, int index)
{
    try
    {
        return array[index];
    }
    catch (System.IndexOutOfRangeException ex)
    {
        Console.WriteLine("Index is out of range: {0}", index);
        throw;
    }
}
```

- Simplify your code by using the C# `using statement`. If you have a `try-finally` statement in which the only

code in the `finally` block is a call to the `Dispose` method, use a `using` statement instead.

In the following example, the `try - finally` statement only calls `Dispose` in the `finally` block.

```
Font font1 = new Font("Arial", 10.0f);
try
{
    byte charset = font1.GdiCharSet;
}
finally
{
    if (font1 != null)
    {
        ((IDisposable)font1).Dispose();
    }
}
```

You can do the same thing with a `using` statement.

```
using (Font font2 = new Font("Arial", 10.0f))
{
    byte charset2 = font2.GdiCharSet;
}
```

In C# 8 and later versions, use the new `using` syntax that doesn't require braces:

```
using Font font3 = new Font("Arial", 10.0f);
byte charset3 = font3.GdiCharSet;
```

`&&` and `||` operators

To avoid exceptions and increase performance by skipping unnecessary comparisons, use `&&` instead of `&` and `||` instead of `|` when you perform comparisons, as shown in the following example.

```
Console.Write("Enter a dividend: ");
int dividend = Convert.ToInt32(Console.ReadLine());

Console.Write("Enter a divisor: ");
int divisor = Convert.ToInt32(Console.ReadLine());

if ((divisor != 0) && (dividend / divisor > 0))
{
    Console.WriteLine("Quotient: {0}", dividend / divisor);
}
else
{
    Console.WriteLine("Attempted division by 0 ends up here.");
}
```

If the divisor is 0, the second clause in the `if` statement would cause a run-time error. But the `&&` operator short-circuits when the first expression is false. That is, it doesn't evaluate the second expression. The `&` operator would evaluate both, resulting in a run-time error when `divisor` is 0.

`new` operator

- Use one of the concise forms of object instantiation, as shown in the following declarations. The second example shows syntax that is available starting in C# 9.

```
var instance1 = new ExampleClass();
```

```
ExampleClass instance2 = new();
```

The preceding declarations are equivalent to the following declaration.

```
ExampleClass instance2 = new ExampleClass();
```

- Use object initializers to simplify object creation, as shown in the following example.

```
var instance3 = new ExampleClass { Name = "Desktop", ID = 37414,  
    Location = "Redmond", Age = 2.3 };
```

The following example sets the same properties as the preceding example but doesn't use initializers.

```
var instance4 = new ExampleClass();  
instance4.Name = "Desktop";  
instance4.ID = 37414;  
instance4.Location = "Redmond";  
instance4.Age = 2.3;
```

Event handling

If you're defining an event handler that you don't need to remove later, use a lambda expression.

```
public Form2()  
{  
    this.Click += (s, e) =>  
    {  
        MessageBox.Show(  
            ((MouseEventArgs)e).Location.ToString());  
    };  
}
```

The lambda expression shortens the following traditional definition.

```
public Form1()  
{  
    this.Click += new EventHandler(Form1_Click);  
}  
  
void Form1_Click(object sender, EventArgs e)  
{  
    MessageBox.Show(((MouseEventArgs)e).Location.ToString());  
}
```

Static members

Call [static](#) members by using the class name: *ClassName.StaticMember*. This practice makes code more readable by making static access clear. Don't qualify a static member defined in a base class with the name of a derived class. While that code compiles, the code readability is misleading, and the code may break in the future if you add a static member with the same name to the derived class.

LINQ queries

- Use meaningful names for query variables. The following example uses `seattleCustomers` for customers

who are located in Seattle.

```
var seattleCustomers = from customer in customers
                        where customer.City == "Seattle"
                        select customer.Name;
```

- Use aliases to make sure that property names of anonymous types are correctly capitalized, using Pascal casing.

```
var localDistributors =
    from customer in customers
    join distributor in distributors on customer.City equals distributor.City
    select new { Customer = customer, Distributor = distributor };
```

- Rename properties when the property names in the result would be ambiguous. For example, if your query returns a customer name and a distributor ID, instead of leaving them as `Name` and `ID` in the result, rename them to clarify that `Name` is the name of a customer, and `ID` is the ID of a distributor.

```
var localDistributors2 =
    from customer in customers
    join distributor in distributors on customer.City equals distributor.City
    select new { CustomerName = customer.Name, DistributorID = distributor.ID };
```

- Use implicit typing in the declaration of query variables and range variables.

```
var seattleCustomers = from customer in customers
                        where customer.City == "Seattle"
                        select customer.Name;
```

- Align query clauses under the `from` clause, as shown in the previous examples.
- Use `where` clauses before other query clauses to ensure that later query clauses operate on the reduced, filtered set of data.

```
var seattleCustomers2 = from customer in customers
                        where customer.City == "Seattle"
                        orderby customer.Name
                        select customer;
```

- Use multiple `from` clauses instead of a `join` clause to access inner collections. For example, a collection of `Student` objects might each contain a collection of test scores. When the following query is executed, it returns each score that is over 90, along with the last name of the student who received the score.

```
var scoreQuery = from student in students
                  from score in student.Scores
                  where score > 90
                  select new { Last = student.LastName, score };
```

Security

Follow the guidelines in [Secure Coding Guidelines](#).

See also

- [.NET runtime coding guidelines](#)
- [Visual Basic Coding Conventions](#)
- [Secure Coding Guidelines](#)

How to display command-line arguments

12/28/2021 • 2 minutes to read • [Edit Online](#)

Arguments provided to an executable on the command line are accessible in [top-level statements](#) or through an optional parameter to `Main`. The arguments are provided in the form of an array of strings. Each element of the array contains one argument. White-space between arguments is removed. For example, consider these command-line invocations of a fictitious executable:

INPUT ON COMMAND LINE	ARRAY OF STRINGS PASSED TO MAIN
<code>executable.exe a b c</code>	<code>"a"</code> <code>"b"</code> <code>"c"</code>
<code>executable.exe one two</code>	<code>"one"</code> <code>"two"</code>
<code>executable.exe "one two" three</code>	<code>"one two"</code> <code>"three"</code>

NOTE

When you are running an application in Visual Studio, you can specify command-line arguments in the [Debug Page, Project Designer](#).

Example

This example displays the command-line arguments passed to a command-line application. The output shown is for the first entry in the table above.


```
using System;

class CommandLine
{
    static void Main(string[] args)
    {
        // The Length property provides the number of array elements.
        Console.WriteLine($"parameter count = {args.Length}");

        for (int i = 0; i < args.Length; i++)
        {
            Console.WriteLine($"Arg[{i}] = [{args[i]}]");
        }
    }
}

/* Output (assumes 3 cmd line args):
   parameter count = 3
   Arg[0] = [a]
   Arg[1] = [b]
   Arg[2] = [c]
*/
```

Explore object oriented programming with classes and objects

12/28/2021 • 9 minutes to read • [Edit Online](#)

In this tutorial, you'll build a console application and see the basic object-oriented features that are part of the C# language.

Prerequisites

The tutorial expects that you have a machine set up for local development. On Windows, Linux, or macOS, you can use the .NET CLI to create, build, and run applications. On Windows, you can use Visual Studio 2019. For setup instructions, see [Set up your local environment](#).

Create your application

Using a terminal window, create a directory named *classes*. You'll build your application there. Change to that directory and type `dotnet new console` in the console window. This command creates your application. Open *Program.cs*. It should look like this:

```
using System;

namespace classes
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Hello World!");
        }
    }
}
```

In this tutorial, you're going to create new types that represent a bank account. Typically developers define each class in a different text file. That makes it easier to manage as a program grows in size. Create a new file named *BankAccount.cs* in the *classes* directory.

This file will contain the definition of a **bank account**. Object Oriented programming organizes code by creating types in the form of *classes*. These classes contain the code that represents a specific entity. The `BankAccount` class represents a bank account. The code implements specific operations through methods and properties. In this tutorial, the bank account supports this behavior:

1. It has a 10-digit number that uniquely identifies the bank account.
2. It has a string that stores the name or names of the owners.
3. The balance can be retrieved.
4. It accepts deposits.
5. It accepts withdrawals.
6. The initial balance must be positive.
7. Withdrawals cannot result in a negative balance.

Define the bank account type

You can start by creating the basics of a class that defines that behavior. Create a new file using the **File:New** command. Name it *BankAccount.cs*. Add the following code to your *BankAccount.cs* file:

```
using System;

namespace classes
{
    public class BankAccount
    {
        public string Number { get; }
        public string Owner { get; set; }
        public decimal Balance { get; }

        public void MakeDeposit(decimal amount, DateTime date, string note)
        {
        }

        public void MakeWithdrawal(decimal amount, DateTime date, string note)
        {
        }
    }
}
```

Before going on, let's take a look at what you've built. The `namespace` declaration provides a way to logically organize your code. This tutorial is relatively small, so you'll put all the code in one namespace.

`public class BankAccount` defines the class, or type, you are creating. Everything inside the `{` and `}` that follows the class declaration defines the state and behavior of the class. There are five *members* of the `BankAccount` class. The first three are *properties*. Properties are data elements and can have code that enforces validation or other rules. The last two are *methods*. Methods are blocks of code that perform a single function. Reading the names of each of the members should provide enough information for you or another developer to understand what the class does.

Open a new account

The first feature to implement is to open a bank account. When a customer opens an account, they must supply an initial balance, and information about the owner or owners of that account.

Creating a new object of the `BankAccount` type means defining a *constructor* that assigns those values. A *constructor* is a member that has the same name as the class. It is used to initialize objects of that class type. Add the following constructor to the `BankAccount` type. Place the following code above the declaration of `MakeDeposit`:

```
public BankAccount(string name, decimal initialBalance)
{
    this.Owner = name;
    this.Balance = initialBalance;
}
```

Constructors are called when you create an object using `new`. Replace the line `Console.WriteLine("Hello World!");` in *Program.cs* with the following code (replace `<name>` with your name):

```
var account = new BankAccount("<name>", 1000);
Console.WriteLine($"Account {account.Number} was created for {account.Owner} with {account.Balance} initial balance.");
```

Let's run what you've built so far. If you're using Visual Studio, Select **Start without debugging** from the

Debug menu. If you're using a command line, type `dotnet run` in the directory where you've created your project.

Did you notice that the account number is blank? It's time to fix that. The account number should be assigned when the object is constructed. But it shouldn't be the responsibility of the caller to create it. The `BankAccount` class code should know how to assign new account numbers. A simple way to do this is to start with a 10-digit number. Increment it when each new account is created. Finally, store the current account number when an object is constructed.

Add a member declaration to the `BankAccount` class. Place the following line of code after the opening brace `{` at the beginning of the `BankAccount` class:

```
private static int accountNumberSeed = 1234567890;
```

This is a data member. It's `private`, which means it can only be accessed by code inside the `BankAccount` class. It's a way of separating the public responsibilities (like having an account number) from the private implementation (how account numbers are generated). It is also `static`, which means it is shared by all of the `BankAccount` objects. The value of a non-static variable is unique to each instance of the `BankAccount` object. Add the following two lines to the constructor to assign the account number. Place them after the line that says

```
this.Balance = initialBalance;
```

```
this.Number = accountNumberSeed.ToString();  
accountNumberSeed++;
```

Type `dotnet run` to see the results.

Create deposits and withdrawals

Your bank account class needs to accept deposits and withdrawals to work correctly. Let's implement deposits and withdrawals by creating a journal of every transaction for the account. That has a few advantages over simply updating the balance on each transaction. The history can be used to audit all transactions and manage daily balances. By computing the balance from the history of all transactions when needed, any errors in a single transaction that are fixed will be correctly reflected in the balance on the next computation.

Let's start by creating a new type to represent a transaction. This is a simple type that doesn't have any responsibilities. It needs a few properties. Create a new file named *Transaction.cs*. Add the following code to it:

```
using System;  
  
namespace classes  
{  
    public class Transaction  
    {  
        public decimal Amount { get; }  
        public DateTime Date { get; }  
        public string Notes { get; }  
  
        public Transaction(decimal amount, DateTime date, string note)  
        {  
            this.Amount = amount;  
            this.Date = date;  
            this.Notes = note;  
        }  
    }  
}
```

Now, let's add a `List<T>` of `Transaction` objects to the `BankAccount` class. Add the following declaration after the constructor in your `BankAccount.cs` file:

```
private List<Transaction> allTransactions = new List<Transaction>();
```

The `List<T>` class requires you to import a different namespace. Add the following at the beginning of `BankAccount.cs`:

```
using System.Collections.Generic;
```

Now, let's correctly compute the `Balance`. The current balance can be found by summing the values of all transactions. As the code is currently, you can only get the initial balance of the account, so you'll have to update the `Balance` property. Replace the line `public decimal Balance { get; }` in `BankAccount.cs` with the following code:

```
public decimal Balance
{
    get
    {
        decimal balance = 0;
        foreach (var item in allTransactions)
        {
            balance += item.Amount;
        }

        return balance;
    }
}
```

This example shows an important aspect of *properties*. You're now computing the balance when another programmer asks for the value. Your computation enumerates all transactions, and provides the sum as the current balance.

Next, implement the `MakeDeposit` and `MakeWithdrawal` methods. These methods will enforce the final two rules: that the initial balance must be positive, and that any withdrawal must not create a negative balance.

This introduces the concept of *exceptions*. The standard way of indicating that a method cannot complete its work successfully is to throw an exception. The type of exception and the message associated with it describe the error. Here, the `MakeDeposit` method throws an exception if the amount of the deposit is not greater than 0. The `MakeWithdrawal` method throws an exception if the withdrawal amount is not greater than 0, or if applying the withdrawal results in a negative balance. Add the following code after the declaration of the `allTransactions` list:

```

public void MakeDeposit(decimal amount, DateTime date, string note)
{
    if (amount <= 0)
    {
        throw new ArgumentOutOfRangeException(nameof(amount), "Amount of deposit must be positive");
    }
    var deposit = new Transaction(amount, date, note);
    allTransactions.Add(deposit);
}

public void MakeWithdrawal(decimal amount, DateTime date, string note)
{
    if (amount <= 0)
    {
        throw new ArgumentOutOfRangeException(nameof(amount), "Amount of withdrawal must be positive");
    }
    if (Balance - amount < 0)
    {
        throw new InvalidOperationException("Not sufficient funds for this withdrawal");
    }
    var withdrawal = new Transaction(-amount, date, note);
    allTransactions.Add(withdrawal);
}

```

The `throw` statement **throws** an exception. Execution of the current block ends, and control transfers to the first matching `catch` block found in the call stack. You'll add a `catch` block to test this code a little later on.

The constructor should get one change so that it adds an initial transaction, rather than updating the balance directly. Since you already wrote the `MakeDeposit` method, call it from your constructor. The finished constructor should look like this:

```

public BankAccount(string name, decimal initialBalance)
{
    this.Number = accountNumberSeed.ToString();
    accountNumberSeed++;

    this.Owner = name;
    MakeDeposit(initialBalance, DateTime.Now, "Initial balance");
}

```

`DateTime.Now` is a property that returns the current date and time. Test this by adding a few deposits and withdrawals in your `Main` method, following the code that creates a new `BankAccount`:

```

account.MakeWithdrawal(500, DateTime.Now, "Rent payment");
Console.WriteLine(account.Balance);
account.MakeDeposit(100, DateTime.Now, "Friend paid me back");
Console.WriteLine(account.Balance);

```

Next, test that you are catching error conditions by trying to create an account with a negative balance. Add the following code after the preceding code you just added:

```
// Test that the initial balances must be positive.
BankAccount invalidAccount;
try
{
    invalidAccount = new BankAccount("invalid", -55);
}
catch (ArgumentOutOfRangeException e)
{
    Console.WriteLine("Exception caught creating account with negative balance");
    Console.WriteLine(e.ToString());
    return;
}
```

You use the `try` and `catch` statements to mark a block of code that may throw exceptions and to catch those errors that you expect. You can use the same technique to test the code that throws an exception for a negative balance. Add the following code at the end of your `Main` method:

```
// Test for a negative balance.
try
{
    account.MakeWithdrawal(750, DateTime.Now, "Attempt to overdraw");
}
catch (InvalidOperationException e)
{
    Console.WriteLine("Exception caught trying to overdraw");
    Console.WriteLine(e.ToString());
}
```

Save the file and type `dotnet run` to try it.

Challenge - log all transactions

To finish this tutorial, you can write the `GetAccountHistory` method that creates a `string` for the transaction history. Add this method to the `BankAccount` type:

```
public string GetAccountHistory()
{
    var report = new System.Text.StringBuilder();

    decimal balance = 0;
    report.AppendLine("Date\t\tAmount\tBalance\tNote");
    foreach (var item in allTransactions)
    {
        balance += item.Amount;
        report.AppendLine($"{item.Date.ToShortDateString()}\t\t{item.Amount}\t\t{balance}\t\t{item.Notes}");
    }

    return report.ToString();
}
```

This uses the `StringBuilder` class to format a string that contains one line for each transaction. You've seen the string formatting code earlier in these tutorials. One new character is `\t`. That inserts a tab to format the output.

Add this line to test it in *Program.cs*:

```
Console.WriteLine(account.GetAccountHistory());
```

Run your program to see the results.

Next steps

If you got stuck, you can see the source for this tutorial [in our GitHub repo](#).

You can continue with the [object oriented programming](#) tutorial.

You can learn more about these concepts in these articles:

- [Selection statements](#)
- [Iteration statements](#)

Object-Oriented programming (C#)

12/28/2021 • 11 minutes to read • [Edit Online](#)

C# is an object-oriented programming language. The four basic principles of object-oriented programming are:

- *Abstraction* Modeling the relevant attributes and interactions of entities as classes to define an abstract representation of a system.
- *Encapsulation* Hiding the internal state and functionality of an object and only allowing access through a public set of functions.
- *Inheritance* Ability to create new abstractions based on existing abstractions.
- *Polymorphism* Ability to implement inherited properties or methods in different ways across multiple abstractions.

In the preceding tutorial, [introduction to classes](#) you saw both *abstraction* and *encapsulation*. The `BankAccount` class provided an abstraction for the concept of a bank account. You could modify its implementation without affecting any of the code that used the `BankAccount` class. Both the `BankAccount` and `Transaction` classes provide encapsulation of the components needed to describe those concepts in code.

In this tutorial, you'll extend that application to make use of *inheritance* and *polymorphism* to add new features. You'll also add features to the `BankAccount` class, taking advantage of the *abstraction* and *encapsulation* techniques you learned in the preceding tutorial.

Create different types of accounts

After building this program, you get requests to add features to it. It works great in the situation where there is only one bank account type. Over time, needs change, and related account types are requested:

- An interest earning account that accrues interest at the end of each month.
- A line of credit that can have a negative balance, but when there's a balance, there's an interest charge each month.
- A pre-paid gift card account that starts with a single deposit, and only can be paid off. It can be refilled once at the start of each month.

All of these different accounts are similar to `BankAccount` class defined in the earlier tutorial. You could copy that code, rename the classes, and make modifications. That technique would work in the short term, but it would be more work over time. Any changes would be copied across all the affected classes.

Instead, you can create new bank account types that inherit methods and data from the `BankAccount` class created in the preceding tutorial. These new classes can extend the `BankAccount` class with the specific behavior needed for each type:

```
public class InterestEarningAccount : BankAccount
{
}

public class LineOfCreditAccount : BankAccount
{
}

public class GiftCardAccount : BankAccount
{
}
```

Each of these classes *inherits* the shared behavior from their shared *base class*, the `BankAccount` class. Write the implementations for new and different functionality in each of the *derived classes*. These derived classes already have all the behavior defined in the `BankAccount` class.

It's a good practice to create each new class in a different source file. In [Visual Studio](#), you can right-click on the project, and select *add class* to add a new class in a new file. In [Visual Studio Code](#), select *File* then *New* to create a new source file. In either tool, name the file to match the class: *InterestEarningAccount.cs*, *LineOfCreditAccount.cs*, and *GiftCardAccount.cs*.

When you create the classes as shown in the preceding sample, you'll find that none of your derived classes compile. A constructor is responsible for initializing an object. A derived class constructor must initialize the derived class, and provide instructions on how to initialize the base class object included in the derived class. The proper initialization normally happens without any extra code. The `BankAccount` class declares one public constructor with the following signature:

```
public BankAccount(string name, decimal initialBalance)
```

The compiler doesn't generate a default constructor when you define a constructor yourself. That means each derived class must explicitly call this constructor. You declare a constructor that can pass arguments to the base class constructor. The following code shows the constructor for the `InterestEarningAccount`:

```
public InterestEarningAccount(string name, decimal initialBalance) : base(name, initialBalance)
{
}
```

The parameters to this new constructor match the parameter type and names of the base class constructor. You use the `: base()` syntax to indicate a call to a base class constructor. Some classes define multiple constructors, and this syntax enables you to pick which base class constructor you call. Once you've updated the constructors, you can develop the code for each of the derived classes. The requirements for the new classes can be stated as follows:

- An interest earning account:
 - Will get a credit of 2% of the month-ending-balance.
- A line of credit:
 - Can have a negative balance, but not be greater in absolute value than the credit limit.
 - Will incur an interest charge each month where the end of month balance isn't 0.
 - Will incur a fee on each withdrawal that goes over the credit limit.
- A gift card account:
 - Can be refilled with a specified amount once each month, on the last day of the month.

You can see that all three of these account types have an action that takes places at the end of each month. However, each account type does different tasks. You use *polymorphism* to implement this code. Create a single `virtual` method in the `BankAccount` class:

```
public virtual void PerformMonthEndTransactions() { }
```

The preceding code shows how you use the `virtual` keyword to declare a method in the base class that a derived class may provide a different implementation for. A `virtual` method is a method where any derived class may choose to reimplement. The derived classes use the `override` keyword to define the new implementation. Typically you refer to this as "overriding the base class implementation". The `virtual` keyword specifies that derived classes may override the behavior. You can also declare `abstract` methods where derived

classes must override the behavior. The base class does not provide an implementation for an `abstract` method. Next, you need to define the implementation for two of the new classes you've created. Start with the `InterestEarningAccount`:

```
public override void PerformMonthEndTransactions()
{
    if (Balance > 500m)
    {
        var interest = Balance * 0.05m;
        MakeDeposit(interest, DateTime.Now, "apply monthly interest");
    }
}
```

Add the following code to the `LineOfCreditAccount`. The code negates the balance to compute a positive interest charge that is withdrawn from the account:

```
public override void PerformMonthEndTransactions()
{
    if (Balance < 0)
    {
        // Negate the balance to get a positive interest charge:
        var interest = -Balance * 0.07m;
        MakeWithdrawal(interest, DateTime.Now, "Charge monthly interest");
    }
}
```

The `GiftCardAccount` class needs two changes to implement its month-end functionality. First, modify the constructor to include an optional amount to add each month:

```
private decimal _monthlyDeposit = 0m;

public GiftCardAccount(string name, decimal initialBalance, decimal monthlyDeposit = 0) : base(name,
initialBalance)
    => _monthlyDeposit = monthlyDeposit;
```

The constructor provides a default value for the `monthlyDeposit` value so callers can omit a `0` for no monthly deposit. Next, override the `PerformMonthEndTransactions` method to add the monthly deposit, if it was set to a non-zero value in the constructor:

```
public override void PerformMonthEndTransactions()
{
    if (_monthlyDeposit != 0)
    {
        MakeDeposit(_monthlyDeposit, DateTime.Now, "Add monthly deposit");
    }
}
```

The override applies the monthly deposit set in the constructor. Add the following code to the `Main` method to test these changes for the `GiftCardAccount` and the `InterestEarningAccount`:

```

var giftCard = new GiftCardAccount("gift card", 100, 50);
giftCard.MakeWithdrawal(20, DateTime.Now, "get expensive coffee");
giftCard.MakeWithdrawal(50, DateTime.Now, "buy groceries");
giftCard.PerformMonthEndTransactions();
// can make additional deposits:
giftCard.MakeDeposit(27.50m, DateTime.Now, "add some additional spending money");
Console.WriteLine(giftCard.GetAccountHistory());

var savings = new InterestEarningAccount("savings account", 10000);
savings.MakeDeposit(750, DateTime.Now, "save some money");
savings.MakeDeposit(1250, DateTime.Now, "Add more savings");
savings.MakeWithdrawal(250, DateTime.Now, "Needed to pay monthly bills");
savings.PerformMonthEndTransactions();
Console.WriteLine(savings.GetAccountHistory());

```

Verify the results. Now, add a similar set of test code for the `LineOfCreditAccount`:

```

var lineOfCredit = new LineOfCreditAccount("line of credit", 0);
// How much is too much to borrow?
lineOfCredit.MakeWithdrawal(1000m, DateTime.Now, "Take out monthly advance");
lineOfCredit.MakeDeposit(50m, DateTime.Now, "Pay back small amount");
lineOfCredit.MakeWithdrawal(5000m, DateTime.Now, "Emergency funds for repairs");
lineOfCredit.MakeDeposit(150m, DateTime.Now, "Partial restoration on repairs");
lineOfCredit.PerformMonthEndTransactions();
Console.WriteLine(lineOfCredit.GetAccountHistory());

```

When you add the preceding code and run the program, you'll see something like the following error:

```

Unhandled exception. System.ArgumentOutOfRangeException: Amount of deposit must be positive (Parameter
'amount')
   at OOProgramming.BankAccount.MakeDeposit(Decimal amount, DateTime date, String note) in
BankAccount.cs:line 42
   at OOProgramming.BankAccount..ctor(String name, Decimal initialBalance) in BankAccount.cs:line 31
   at OOProgramming.LineOfCreditAccount..ctor(String name, Decimal initialBalance) in
LineOfCreditAccount.cs:line 9
   at OOProgramming.Program.Main(String[] args) in Program.cs:line 29

```

NOTE

The actual output includes the full path to the folder with the project. The folder names were omitted for brevity. Also, depending on your code format, the line numbers may be slightly different.

This code fails because the `BankAccount` assumes that the initial balance must be greater than 0. Another assumption baked into the `BankAccount` class is that the balance can't go negative. Instead, any withdrawal that overdraws the account is rejected. Both of those assumptions need to change. The line of credit account starts at 0, and generally will have a negative balance. Also, if a customer borrows too much money, they incur a fee. The transaction is accepted, it just costs more. The first rule can be implemented by adding an optional argument to the `BankAccount` constructor that specifies the minimum balance. The default is `0`. The second rule requires a mechanism that enables derived classes to modify the default algorithm. In a sense, the base class "asks" the derived type what should happen when there's an overdraft. The default behavior is to reject the transaction by throwing an exception.

Let's start by adding a second constructor that includes an optional `minimumBalance` parameter. This new constructor does all the actions done by the existing constructor. Also, it sets the minimum balance property. You could copy the body of the existing constructor, but that means two locations to change in the future. Instead, you can use *constructor chaining* to have one constructor call another. The following code shows the two

constructors and the new additional field:

```
private readonly decimal minimumBalance;

public BankAccount(string name, decimal initialBalance) : this(name, initialBalance, 0) { }

public BankAccount(string name, decimal initialBalance, decimal minimumBalance)
{
    this.Number = accountNumberSeed.ToString();
    accountNumberSeed++;

    this.Owner = name;
    this.minimumBalance = minimumBalance;
    if (initialBalance > 0)
        MakeDeposit(initialBalance, DateTime.Now, "Initial balance");
}
```

The preceding code shows two new techniques. First, the `minimumBalance` field is marked as `readonly`. That means the value cannot be changed after the object is constructed. Once a `BankAccount` is created, the `minimumBalance` can't change. Second, the constructor that takes two parameters uses `: this(name, initialBalance, 0) { }` as its implementation. The `: this()` expression calls the other constructor, the one with three parameters. This technique allows you to have a single implementation for initializing an object even though client code can choose one of many constructors.

This implementation calls `MakeDeposit` only if the initial balance is greater than `0`. That preserves the rule that deposits must be positive, yet lets the credit account open with a `0` balance.

Now that the `BankAccount` class has a read-only field for the minimum balance, the final change is to change the hard code `0` to `minimumBalance` in the `MakeWithdrawal` method:

```
if (Balance - amount < minimumBalance)
```

After extending the `BankAccount` class, you can modify the `LineOfCreditAccount` constructor to call the new base constructor, as shown in the following code:

```
public LineOfCreditAccount(string name, decimal initialBalance, decimal creditLimit) : base(name,
initialBalance, -creditLimit)
{
}
```

Notice that the `LineOfCreditAccount` constructor changes the sign of the `creditLimit` parameter so it matches the meaning of the `minimumBalance` parameter.

Different overdraft rules

The last feature to add enables the `LineOfCreditAccount` to charge a fee for going over the credit limit instead of refusing the transaction.

One technique is to define a virtual function where you implement the required behavior. The `BankAccount` class refactors the `MakeWithdrawal` method into two methods. The new method does the specified action when the withdrawal takes the balance below the minimum. The existing `MakeWithdrawal` method has the following code:

```

public void MakeWithdrawal(decimal amount, DateTime date, string note)
{
    if (amount <= 0)
    {
        throw new ArgumentOutOfRangeException(nameof(amount), "Amount of withdrawal must be positive");
    }
    if (Balance - amount < minimumBalance)
    {
        throw new InvalidOperationException("Not sufficient funds for this withdrawal");
    }
    var withdrawal = new Transaction(-amount, date, note);
    allTransactions.Add(withdrawal);
}

```

Replace it with the following code:

```

public void MakeWithdrawal(decimal amount, DateTime date, string note)
{
    if (amount <= 0)
    {
        throw new ArgumentOutOfRangeException(nameof(amount), "Amount of withdrawal must be positive");
    }
    var overdraftTransaction = CheckWithdrawalLimit(Balance - amount < minimumBalance);
    var withdrawal = new Transaction(-amount, date, note);
    allTransactions.Add(withdrawal);
    if (overdraftTransaction != null)
        allTransactions.Add(overdraftTransaction);
}

protected virtual Transaction? CheckWithdrawalLimit(bool isOverdrawn)
{
    if (isOverdrawn)
    {
        throw new InvalidOperationException("Not sufficient funds for this withdrawal");
    }
    else
    {
        return default;
    }
}

```

The added method is `protected`, which means that it can be called only from derived classes. That declaration prevents other clients from calling the method. It's also `virtual` so that derived classes can change the behavior. The return type is a `Transaction?`. The `?` annotation indicates that the method may return `null`. Add the following implementation in the `LineOfCreditAccount` to charge a fee when the withdrawal limit is exceeded:

```

protected override Transaction? CheckWithdrawalLimit(bool isOverdrawn) =>
    isOverdrawn
    ? new Transaction(-20, DateTime.Now, "Apply overdraft fee")
    : default;

```

The override returns a fee transaction when the account is overdrawn. If the withdrawal doesn't go over the limit, the method returns a `null` transaction. That indicates there's no fee. Test these changes by adding the following code to your `Main` method in the `Program` class:

```
var lineOfCredit = new LineOfCreditAccount("line of credit", 0, 2000);
// How much is too much to borrow?
lineOfCredit.MakeWithdrawal(1000m, DateTime.Now, "Take out monthly advance");
lineOfCredit.MakeDeposit(50m, DateTime.Now, "Pay back small amount");
lineOfCredit.MakeWithdrawal(5000m, DateTime.Now, "Emergency funds for repairs");
lineOfCredit.MakeDeposit(150m, DateTime.Now, "Partial restoration on repairs");
lineOfCredit.PerformMonthEndTransactions();
Console.WriteLine(lineOfCredit.GetAccountHistory());
```

Run the program, and check the results.

Summary

If you got stuck, you can see the source for this tutorial [in our GitHub repo](#).

This tutorial demonstrated many of the techniques used in Object-Oriented programming:

- You used *Abstraction* when you defined classes for each of the different account types. Those classes described the behavior for that type of account.
- You used *Encapsulation* when you kept many details `private` in each class.
- You used *Inheritance* when you leveraged the implementation already created in the `BankAccount` class to save code.
- You used *Polymorphism* when you created `virtual` methods that derived classes could override to create specific behavior for that account type.

Inheritance in C# and .NET

12/28/2021 • 25 minutes to read • [Edit Online](#)

This tutorial introduces you to inheritance in C#. Inheritance is a feature of object-oriented programming languages that allows you to define a base class that provides specific functionality (data and behavior) and to define derived classes that either inherit or override that functionality.

Prerequisites

This tutorial assumes that you've installed the .NET SDK. Visit the [.NET Downloads](#) page to download it. You also need a code editor. This tutorial uses [Visual Studio Code](#), although you can use any code editor of your choice.

Running the examples

To create and run the examples in this tutorial, you use the [dotnet](#) utility from the command line. Follow these steps for each example:

1. Create a directory to store the example.
2. Enter the [dotnet new console](#) command at a command prompt to create a new .NET Core project.
3. Copy and paste the code from the example into your code editor.
4. Enter the [dotnet restore](#) command from the command line to load or restore the project's dependencies.

You don't have to run `dotnet restore` because it's run implicitly by all commands that require a restore to occur, such as `dotnet new`, `dotnet build`, `dotnet run`, `dotnet test`, `dotnet publish`, and `dotnet pack`. To disable implicit restore, use the `--no-restore` option.

The `dotnet restore` command is still useful in certain scenarios where explicitly restoring makes sense, such as [continuous integration builds in Azure DevOps Services](#) or in build systems that need to explicitly control when the restore occurs.

For information about how to manage NuGet feeds, see the [dotnet restore](#) [documentation](#).

5. Enter the [dotnet run](#) command to compile and execute the example.

Background: What is inheritance?

Inheritance is one of the fundamental attributes of object-oriented programming. It allows you to define a child class that reuses (inherits), extends, or modifies the behavior of a parent class. The class whose members are inherited is called the *base class*. The class that inherits the members of the base class is called the *derived class*.

C# and .NET support *single inheritance* only. That is, a class can only inherit from a single class. However, inheritance is transitive, which allows you to define an inheritance hierarchy for a set of types. In other words, type `D` can inherit from type `C`, which inherits from type `B`, which inherits from the base class type `A`. Because inheritance is transitive, the members of type `A` are available to type `D`.

Not all members of a base class are inherited by derived classes. The following members are not inherited:

- [Static constructors](#), which initialize the static data of a class.
- [Instance constructors](#), which you call to create a new instance of the class. Each class must define its own constructors.

- **Finalizers**, which are called by the runtime's garbage collector to destroy instances of a class.

While all other members of a base class are inherited by derived classes, whether they are visible or not depends on their accessibility. A member's accessibility affects its visibility for derived classes as follows:

- **Private** members are visible only in derived classes that are nested in their base class. Otherwise, they are not visible in derived classes. In the following example, `A.B` is a nested class that derives from `A`, and `C` derives from `A`. The private `A.value` field is visible in `A.B`. However, if you remove the comments from the `C.GetValue` method and attempt to compile the example, it produces compiler error CS0122: "'A.value' is inaccessible due to its protection level."

```
using System;

public class A
{
    private int value = 10;

    public class B : A
    {
        public int GetValue()
        {
            return this.value;
        }
    }
}

public class C : A
{
    // public int GetValue()
    // {
    //     return this.value;
    // }
}

public class AccessExample
{
    public static void Main(string[] args)
    {
        var b = new A.B();
        Console.WriteLine(b.GetValue());
    }
}

// The example displays the following output:
//      10
```

- **Protected** members are visible only in derived classes.
- **Internal** members are visible only in derived classes that are located in the same assembly as the base class. They are not visible in derived classes located in a different assembly from the base class.
- **Public** members are visible in derived classes and are part of the derived class' public interface. Public inherited members can be called just as if they are defined in the derived class. In the following example, class `A` defines a method named `Method1`, and class `B` inherits from class `A`. The example then calls `Method1` as if it were an instance method on `B`.

```

public class A
{
    public void Method1()
    {
        // Method implementation.
    }
}

public class B : A
{ }

public class Example
{
    public static void Main()
    {
        B b = new B();
        b.Method1();
    }
}

```

Derived classes can also *override* inherited members by providing an alternate implementation. In order to be able to override a member, the member in the base class must be marked with the [virtual](#) keyword. By default, base class members are not marked as `virtual` and cannot be overridden. Attempting to override a non-virtual member, as the following example does, generates compiler error CS0506: "<member> cannot override inherited member <member> because it is not marked virtual, abstract, or override."

```

public class A
{
    public void Method1()
    {
        // Do something.
    }
}

public class B : A
{
    public override void Method1() // Generates CS0506.
    {
        // Do something else.
    }
}

```

In some cases, a derived class *must* override the base class implementation. Base class members marked with the [abstract](#) keyword require that derived classes override them. Attempting to compile the following example generates compiler error CS0534, "<class> does not implement inherited abstract member <member>", because class `B` provides no implementation for `A.Method1`.

```

public abstract class A
{
    public abstract void Method1();
}

public class B : A // Generates CS0534.
{
    public void Method3()
    {
        // Do something.
    }
}

```

Inheritance applies only to classes and interfaces. Other type categories (structs, delegates, and enums) do not support inheritance. Because of these rules, attempting to compile code like the following example produces compiler error CS0527: "Type 'ValueType' in interface list is not an interface." The error message indicates that, although you can define the interfaces that a struct implements, inheritance is not supported.

```
using System;

public struct ValueStructure : ValueType // Generates CS0527.
{
}
```

Implicit inheritance

Besides any types that they may inherit from through single inheritance, all types in the .NET type system implicitly inherit from [Object](#) or a type derived from it. The common functionality of [Object](#) is available to any type.

To see what implicit inheritance means, let's define a new class, `SimpleClass`, that is simply an empty class definition:

```
public class SimpleClass
{ }
```

You can then use reflection (which lets you inspect a type's metadata to get information about that type) to get a list of the members that belong to the `SimpleClass` type. Although you haven't defined any members in your `SimpleClass` class, output from the example indicates that it actually has nine members. One of these members is a parameterless (or default) constructor that is automatically supplied for the `SimpleClass` type by the C# compiler. The remaining eight are members of [Object](#), the type from which all classes and interfaces in the .NET type system ultimately implicitly inherit.

```

using System;
using System.Reflection;

public class SimpleClassExample
{
    public static void Main()
    {
        Type t = typeof(SimpleClass);
        BindingFlags flags = BindingFlags.Instance | BindingFlags.Static | BindingFlags.Public |
            BindingFlags.NonPublic | BindingFlags.FlattenHierarchy;
        MemberInfo[] members = t.GetMembers(flags);
        Console.WriteLine($"Type {t.Name} has {members.Length} members: ");
        foreach (var member in members)
        {
            string access = "";
            string stat = "";
            var method = member as MethodBase;
            if (method != null)
            {
                if (method.IsPublic)
                    access = " Public";
                else if (method.IsPrivate)
                    access = " Private";
                else if (method.IsFamily)
                    access = " Protected";
                else if (method.IsAssembly)
                    access = " Internal";
                else if (method.IsFamilyOrAssembly)
                    access = " Protected Internal ";
                if (method.IsStatic)
                    stat = " Static";
            }
            var output = $"{member.Name} ({member.MemberType}): {access}{stat}, Declared by {member.DeclaringType}";
            Console.WriteLine(output);
        }
    }
}

// The example displays the following output:
// Type SimpleClass has 9 members:
// ToString (Method): Public, Declared by System.Object
// Equals (Method): Public, Declared by System.Object
// Equals (Method): Public Static, Declared by System.Object
// ReferenceEquals (Method): Public Static, Declared by System.Object
// GetHashCode (Method): Public, Declared by System.Object
// GetType (Method): Public, Declared by System.Object
// Finalize (Method): Internal, Declared by System.Object
// MemberwiseClone (Method): Internal, Declared by System.Object
// .ctor (Constructor): Public, Declared by SimpleClass

```

Implicit inheritance from the [Object](#) class makes these methods available to the `SimpleClass` class:

- The public `ToString` method, which converts a `SimpleClass` object to its string representation, returns the fully qualified type name. In this case, the `ToString` method returns the string "SimpleClass".
- Three methods that test for equality of two objects: the public instance `Equals(Object)` method, the public static `Equals(Object, Object)` method, and the public static `ReferenceEquals(Object, Object)` method. By default, these methods test for reference equality; that is, to be equal, two object variables must refer to the same object.
- The public `GetHashCode` method, which computes a value that allows an instance of the type to be used in hashed collections.
- The public `GetType` method, which returns a [Type](#) object that represents the `SimpleClass` type.

- The protected [Finalize](#) method, which is designed to release unmanaged resources before an object's memory is reclaimed by the garbage collector.
- The protected [MemberwiseClone](#) method, which creates a shallow clone of the current object.

Because of implicit inheritance, you can call any inherited member from a `SimpleClass` object just as if it was actually a member defined in the `SimpleClass` class. For instance, the following example calls the `SimpleClass.ToString` method, which `SimpleClass` inherits from [Object](#).

```
using System;

public class EmptyClass
{
}

public class ClassNameExample
{
    public static void Main()
    {
        EmptyClass sc = new EmptyClass();
        Console.WriteLine(sc.ToString());
    }
}
// The example displays the following output:
//      EmptyClass
```

The following table lists the categories of types that you can create in C# and the types from which they implicitly inherit. Each base type makes a different set of members available through inheritance to implicitly derived types.

TYPE CATEGORY	IMPLICITLY INHERITS FROM
class	Object
struct	ValueType , Object
enum	Enum , ValueType , Object
delegate	MulticastDelegate , Delegate , Object

Inheritance and an "is a" relationship

Ordinarily, inheritance is used to express an "is a" relationship between a base class and one or more derived classes, where the derived classes are specialized versions of the base class; the derived class is a type of the base class. For example, the `Publication` class represents a publication of any kind, and the `Book` and `Magazine` classes represent specific types of publications.

NOTE

A class or struct can implement one or more interfaces. While interface implementation is often presented as a workaround for single inheritance or as a way of using inheritance with structs, it is intended to express a different relationship (a "can do" relationship) between an interface and its implementing type than inheritance. An interface defines a subset of functionality (such as the ability to test for equality, to compare or sort objects, or to support culture-sensitive parsing and formatting) that the interface makes available to its implementing types.

Note that "is a" also expresses the relationship between a type and a specific instantiation of that type. In the following example, `Automobile` is a class that has three unique read-only properties: `Make`, the manufacturer of

the automobile; `Model`, the kind of automobile; and `Year`, its year of manufacture. Your `Automobile` class also has a constructor whose arguments are assigned to the property values, and it overrides the [Object.ToString](#) method to produce a string that uniquely identifies the `Automobile` instance rather than the `Automobile` class.

```
using System;

public class Automobile
{
    public Automobile(string make, string model, int year)
    {
        if (make == null)
            throw new ArgumentNullException("The make cannot be null.");
        else if (String.IsNullOrEmpty(make))
            throw new ArgumentException("make cannot be an empty string or have space characters only.");
        Make = make;

        if (model == null)
            throw new ArgumentNullException("The model cannot be null.");
        else if (String.IsNullOrEmpty(model))
            throw new ArgumentException("model cannot be an empty string or have space characters only.");
        Model = model;

        if (year < 1857 || year > DateTime.Now.Year + 2)
            throw new ArgumentException("The year is out of range.");
        Year = year;
    }

    public string Make { get; }

    public string Model { get; }

    public int Year { get; }

    public override string ToString() => $"{Year} {Make} {Model}";
}
```

In this case, you shouldn't rely on inheritance to represent specific car makes and models. For example, you don't need to define a `Packard` type to represent automobiles manufactured by the Packard Motor Car Company. Instead, you can represent them by creating an `Automobile` object with the appropriate values passed to its class constructor, as the following example does.

```
using System;

public class Example
{
    public static void Main()
    {
        var packard = new Automobile("Packard", "Custom Eight", 1948);
        Console.WriteLine(packard);
    }
}

// The example displays the following output:
//      1948 Packard Custom Eight
```

An is-a relationship based on inheritance is best applied to a base class and to derived classes that add additional members to the base class or that require additional functionality not present in the base class.

Designing the base class and derived classes

Let's look at the process of designing a base class and its derived classes. In this section, you'll define a base class, `Publication`, which represents a publication of any kind, such as a book, a magazine, a newspaper, a

journal, an article, etc. You'll also define a `Book` class that derives from `Publication`. You could easily extend the example to define other derived classes, such as `Magazine`, `Journal`, `Newspaper`, and `Article`.

The base `Publication` class

In designing your `Publication` class, you need to make several design decisions:

- What members to include in your base `Publication` class, and whether the `Publication` members provide method implementations or whether `Publication` is an abstract base class that serves as a template for its derived classes.

In this case, the `Publication` class will provide method implementations. The [Designing abstract base classes and their derived classes](#) section contains an example that uses an abstract base class to define the methods that derived classes must override. Derived classes are free to provide any implementation that is suitable for the derived type.

The ability to reuse code (that is, multiple derived classes share the declaration and implementation of base class methods and do not need to override them) is an advantage of non-abstract base classes. Therefore, you should add members to `Publication` if their code is likely to be shared by some or most specialized `Publication` types. If you fail to provide base class implementations efficiently, you'll end up having to provide largely identical member implementations in derived classes rather than a single implementation in the base class. The need to maintain duplicated code in multiple locations is a potential source of bugs.

Both to maximize code reuse and to create a logical and intuitive inheritance hierarchy, you want to be sure that you include in the `Publication` class only the data and functionality that is common to all or to most publications. Derived classes then implement members that are unique to the particular kinds of publication that they represent.

- How far to extend your class hierarchy. Do you want to develop a hierarchy of three or more classes, rather than simply a base class and one or more derived classes? For example, `Publication` could be a base class of `Periodical`, which in turn is a base class of `Magazine`, `Journal` and `Newspaper`.

For your example, you'll use the small hierarchy of a `Publication` class and a single derived class, `Book`. You could easily extend the example to create a number of additional classes that derive from `Publication`, such as `Magazine` and `Article`.

- Whether it makes sense to instantiate the base class. If it does not, you should apply the `abstract` keyword to the class. Otherwise, your `Publication` class can be instantiated by calling its class constructor. If an attempt is made to instantiate a class marked with the `abstract` keyword by a direct call to its class constructor, the C# compiler generates error CS0144, "Cannot create an instance of the abstract class or interface." If an attempt is made to instantiate the class by using reflection, the reflection method throws a [MemberAccessException](#).

By default, a base class can be instantiated by calling its class constructor. You do not have to explicitly define a class constructor. If one is not present in the base class' source code, the C# compiler automatically provides a default (parameterless) constructor.

For your example, you'll mark the `Publication` class as `abstract` so that it cannot be instantiated. An `abstract` class without any `abstract` methods indicates that this class represents an abstract concept that is shared among several concrete classes (like a `Book`, `Journal`).

- Whether derived classes must inherit the base class implementation of particular members, whether they have the option to override the base class implementation, or whether they must provide an implementation. You use the `abstract` keyword to force derived classes to provide an implementation. You use the `virtual` keyword to allow derived classes to override a base class method. By default, methods defined in the base class are *not* overridable.

The `Publication` class does not have any `abstract` methods, but the class itself is `abstract`.

- Whether a derived class represents the final class in the inheritance hierarchy and cannot itself be used as a base class for additional derived classes. By default, any class can serve as a base class. You can apply the `sealed` keyword to indicate that a class cannot serve as a base class for any additional classes. Attempting to derive from a sealed class generated compiler error CS0509, "cannot derive from sealed type <typeName>".

For your example, you'll mark your derived class as `sealed`.

The following example shows the source code for the `Publication` class, as well as a `PublicationType` enumeration that is returned by the `Publication.PublicationType` property. In addition to the members that it inherits from `Object`, the `Publication` class defines the following unique members and member overrides:

```
using System;

public enum PublicationType { Misc, Book, Magazine, Article };

public abstract class Publication
{
    private bool published = false;
    private DateTime datePublished;
    private int totalPages;

    public Publication(string title, string publisher, PublicationType type)
    {
        if (String.IsNullOrEmpty(publisher))
            throw new ArgumentException("The publisher is required.");
        Publisher = publisher;

        if (String.IsNullOrEmpty(title))
            throw new ArgumentException("The title is required.");
        Title = title;

        Type = type;
    }

    public string Publisher { get; }

    public string Title { get; }

    public PublicationType Type { get; }

    public string CopyrightName { get; private set; }

    public int CopyrightDate { get; private set; }

    public int Pages
    {
        get { return totalPages; }
        set
        {
            if (value <= 0)
                throw new ArgumentOutOfRangeException("The number of pages cannot be zero or negative.");
            totalPages = value;
        }
    }

    public string GetPublicationDate()
    {
        if (!published)
            return "NYP";
        else
            return datePublished.ToString("d");
    }
}
```



```

public void Publish(DateTime datePublished)
{
    published = true;
    this.datePublished = datePublished;
}

public void Copyright(string copyrightName, int copyrightDate)
{
    if (String.IsNullOrEmpty(copyrightName))
        throw new ArgumentException("The name of the copyright holder is required.");
    CopyrightName = copyrightName;

    int currentYear = DateTime.Now.Year;
    if (copyrightDate < currentYear - 10 || copyrightDate > currentYear + 2)
        throw new ArgumentOutOfRangeException($"The copyright year must be between {currentYear - 10} and {currentYear + 1}");
    CopyrightDate = copyrightDate;
}

public override string ToString() => Title;
}

```

- A constructor

Because the `Publication` class is `abstract`, it cannot be instantiated directly from code like the following example:

```

var publication = new Publication("Tiddlywinks for Experts", "Fun and Games",
                                PublicationType.Book);

```

However, its instance constructor can be called directly from derived class constructors, as the source code for the `Book` class shows.

- Two publication-related properties

`Title` is a read-only `String` property whose value is supplied by calling the `Publication` constructor.

`Pages` is a read-write `Int32` property that indicates how many total pages the publication has. The value is stored in a private field named `totalPages`. It must be a positive number or an `ArgumentOutOfRangeException` is thrown.

- Publisher-related members

Two read-only properties, `Publisher` and `Type`. The values are originally supplied by the call to the `Publication` class constructor.

- Publishing-related members

Two methods, `Publish` and `GetPublicationDate`, set and return the publication date. The `Publish` method sets a private `published` flag to `true` when it is called and assigns the date passed to it as an argument to the private `datePublished` field. The `GetPublicationDate` method returns the string "NYP" if the `published` flag is `false`, and the value of the `datePublished` field if it is `true`.

- Copyright-related members

The `Copyright` method takes the name of the copyright holder and the year of the copyright as arguments and assigns them to the `CopyrightName` and `CopyrightDate` properties.

- An override of the `ToString` method

If a type does not override the `Object.ToString` method, it returns the fully qualified name of the type,

which is of little use in differentiating one instance from another. The `Publication` class overrides `Object.ToString` to return the value of the `Title` property.

The following figure illustrates the relationship between your base `Publication` class and its implicitly inherited `Object` class.

Object	Publication
Equals(Object)	Equals(Object)
Equals(Object, Object)	Equals(Object, Object)
Finalize()	Finalize()
GetHashCode()	GetHashCode()
GetType()	GetType()
MemberwiseClone()	MemberwiseClone()
ReferenceEquals()	ReferenceEquals()
ToString()	ToString()
#ctor()	#ctor(String, String, PublicationType)
	PublicationType
	Publisher
	Title
	CopyrightDate
	CopyrightName
	Pages
	Copyright()
	GetPublicationDate()
	Publish()

Key

Unique member	
Inherited member	
Overridden member	

The `Book` class

The `Book` class represents a book as a specialized type of publication. The following example shows the source code for the `Book` class.

```

using System;

public sealed class Book : Publication
{
    public Book(string title, string author, string publisher) :
        this(title, String.Empty, author, publisher)
    { }

    public Book(string title, string isbn, string author, string publisher) : base(title, publisher,
PublicationType.Book)
    {
        // isbn argument must be a 10- or 13-character numeric string without "-" characters.
        // We could also determine whether the ISBN is valid by comparing its checksum digit
        // with a computed checksum.
        //
        if (! String.IsNullOrEmpty(isbn)) {
            // Determine if ISBN length is correct.
            if (! (isbn.Length == 10 | isbn.Length == 13))
                throw new ArgumentException("The ISBN must be a 10- or 13-character numeric string.");
            ulong nISBN = 0;
            if (! UInt64.TryParse(isbn, out nISBN))
                throw new ArgumentException("The ISBN can consist of numeric characters only.");
        }
        ISBN = isbn;

        Author = author;
    }

    public string ISBN { get; }

    public string Author { get; }

    public Decimal Price { get; private set; }

    // A three-digit ISO currency symbol.
    public string Currency { get; private set; }

    // Returns the old price, and sets a new price.
    public Decimal SetPrice(Decimal price, string currency)
    {
        if (price < 0)
            throw new ArgumentOutOfRangeException("The price cannot be negative.");
        Decimal oldValue = Price;
        Price = price;

        if (currency.Length != 3)
            throw new ArgumentException("The ISO currency symbol is a 3-character string.");
        Currency = currency;

        return oldValue;
    }

    public override bool Equals(object obj)
    {
        Book book = obj as Book;
        if (book == null)
            return false;
        else
            return ISBN == book.ISBN;
    }

    public override int GetHashCode() => ISBN.GetHashCode();

    public override string ToString() => $"{(String.IsNullOrEmpty(Author) ? "" : Author + ", ")}{Title}";
}

```

In addition to the members that it inherits from `Publication`, the `Book` class defines the following unique members and member overrides:

- Two constructors

The two `Book` constructors share three common parameters. Two, *title* and *publisher*, correspond to parameters of the `Publication` constructor. The third is *author*, which is stored to a public immutable `Author` property. One constructor includes an *isbn* parameter, which is stored in the `ISBN` auto-property.

The first constructor uses the `this` keyword to call the other constructor. Constructor chaining is a common pattern in defining constructors. Constructors with fewer parameters provide default values when calling the constructor with the greatest number of parameters.

The second constructor uses the `base` keyword to pass the title and publisher name to the base class constructor. If you don't make an explicit call to a base class constructor in your source code, the C# compiler automatically supplies a call to the base class' default or parameterless constructor.

- A read-only `ISBN` property, which returns the `Book` object's International Standard Book Number, a unique 10- or 13-digit number. The ISBN is supplied as an argument to one of the `Book` constructors. The ISBN is stored in a private backing field, which is auto-generated by the compiler.
- A read-only `Author` property. The author name is supplied as an argument to both `Book` constructors and is stored in the property.
- Two read-only price-related properties, `Price` and `Currency`. Their values are provided as arguments in a `SetPrice` method call. The `Currency` property is the three-digit ISO currency symbol (for example, USD for the U.S. dollar). ISO currency symbols can be retrieved from the `ISOCurrencySymbol` property. Both of these properties are externally read-only, but both can be set by code in the `Book` class.
- A `SetPrice` method, which sets the values of the `Price` and `Currency` properties. Those values are returned by those same properties.
- Overrides to the `ToString` method (inherited from `Publication`) and the `Object.Equals(Object)` and `GetHashCode` methods (inherited from `Object`).

Unless it is overridden, the `Object.Equals(Object)` method tests for reference equality. That is, two object variables are considered to be equal if they refer to the same object. In the `Book` class, on the other hand, two `Book` objects should be equal if they have the same ISBN.

When you override the `Object.Equals(Object)` method, you must also override the `GetHashCode` method, which returns a value that the runtime uses to store items in hashed collections for efficient retrieval. The hash code should return a value that's consistent with the test for equality. Since you've overridden `Object.Equals(Object)` to return `true` if the ISBN properties of two `Book` objects are equal, you return the hash code computed by calling the `GetHashCode` method of the string returned by the `ISBN` property.

The following figure illustrates the relationship between the `Book` class and `Publication`, its base class.

Publication

Equals(Object)
Equals(Object, Object)
Finalize()
GetHashCode()
GetType()
MemberwiseClone()
ReferenceEquals(Object, Object)
ToString()
#ctor(String, String, PublicationType)
PublicationType
Publisher
Title
CopyrightDate
CopyrightName
Pages
Copyright()
GetPublicationDate()
Publish()

Key

Unique member	
Inherited member	
Overridden member	

Book

Equals(Object)
Equals(Object, Object)
Finalize()
GetHashCode()
GetType()
MemberwiseClone()
ReferenceEquals(Object, Object)
ToString()
#ctor(String, String, String)
#ctor(String, String, String, String)
PublicationType
Publisher
Author
Title
CopyrightDate
CopyrightName
ISBN
Pages
Price
Currency
Copyright()
GetPublicationDate()
Publish()
SetPrice()

You can now instantiate a `Book` object, invoke both its unique and inherited members, and pass it as an argument to a method that expects a parameter of type `Publication` or of type `Book`, as the following example shows.

```

using System;
using static System.Console;

public class ClassExample
{
    public static void Main()
    {
        var book = new Book("The Tempest", "0971655819", "Shakespeare, William",
                             "Public Domain Press");
        ShowPublicationInfo(book);
        book.Publish(new DateTime(2016, 8, 18));
        ShowPublicationInfo(book);

        var book2 = new Book("The Tempest", "Classic Works Press", "Shakespeare, William");
        Write($"{book.Title} and {book2.Title} are the same publication: " +
              $"{((Publication) book).Equals(book2)}");
    }

    public static void ShowPublicationInfo(Publication pub)
    {
        string pubDate = pub.GetPublicationDate();
        WriteLine($"{pub.Title}, " +
                  $"{(pubDate == "NYP" ? "Not Yet Published" : "published on " + pubDate):d} by
{pub.Publisher}");
    }
}
// The example displays the following output:
//      The Tempest, Not Yet Published by Public Domain Press
//      The Tempest, published on 8/18/2016 by Public Domain Press
//      The Tempest and The Tempest are the same publication: False

```

Designing abstract base classes and their derived classes

In the previous example, you defined a base class that provided an implementation for a number of methods to allow derived classes to share code. In many cases, however, the base class is not expected to provide an implementation. Instead, the base class is an *abstract class* that declares *abstract methods*; it serves as a template that defines the members that each derived class must implement. Typically in an abstract base class, the implementation of each derived type is unique to that type. You marked the class with the `abstract` keyword because it made no sense to instantiate a `Publication` object, although the class did provide implementations of functionality common to publications.

For example, each closed two-dimensional geometric shape includes two properties: area, the inner extent of the shape; and perimeter, or the distance along the edges of the shape. The way in which these properties are calculated, however, depends completely on the specific shape. The formula for calculating the perimeter (or circumference) of a circle, for example, is different from that of a triangle. The `Shape` class is an `abstract` class with `abstract` methods. That indicates derived classes share the same functionality, but those derived classes implement that functionality differently.

The following example defines an abstract base class named `Shape` that defines two properties: `Area` and `Perimeter`. In addition to marking the class with the `abstract` keyword, each instance member is also marked with the `abstract` keyword. In this case, `Shape` also overrides the `Object.ToString` method to return the name of the type, rather than its fully qualified name. And it defines two static members, `GetArea` and `GetPerimeter`, that allow callers to easily retrieve the area and perimeter of an instance of any derived class. When you pass an instance of a derived class to either of these methods, the runtime calls the method override of the derived class.

```
using System;

public abstract class Shape
{
    public abstract double Area { get; }

    public abstract double Perimeter { get; }

    public override string ToString() => GetType().Name;

    public static double GetArea(Shape shape) => shape.Area;

    public static double GetPerimeter(Shape shape) => shape.Perimeter;
}
```

You can then derive some classes from `Shape` that represent specific shapes. The following example defines three classes, `Triangle`, `Rectangle`, and `Circle`. Each uses a formula unique for that particular shape to compute the area and perimeter. Some of the derived classes also define properties, such as `Rectangle.Diagonal` and `Circle.Diameter`, that are unique to the shape that they represent.

```

using System;

public class Square : Shape
{
    public Square(double length)
    {
        Side = length;
    }

    public double Side { get; }

    public override double Area => Math.Pow(Side, 2);

    public override double Perimeter => Side * 4;

    public double Diagonal => Math.Round(Math.Sqrt(2) * Side, 2);
}

public class Rectangle : Shape
{
    public Rectangle(double length, double width)
    {
        Length = length;
        Width = width;
    }

    public double Length { get; }

    public double Width { get; }

    public override double Area => Length * Width;

    public override double Perimeter => 2 * Length + 2 * Width;

    public bool IsSquare() => Length == Width;

    public double Diagonal => Math.Round(Math.Sqrt(Math.Pow(Length, 2) + Math.Pow(Width, 2)), 2);
}

public class Circle : Shape
{
    public Circle(double radius)
    {
        Radius = radius;
    }

    public override double Area => Math.Round(Math.PI * Math.Pow(Radius, 2), 2);

    public override double Perimeter => Math.Round(Math.PI * 2 * Radius, 2);

    // Define a circumference, since it's the more familiar term.
    public double Circumference => Perimeter;

    public double Radius { get; }

    public double Diameter => Radius * 2;
}

```

The following example uses objects derived from `Shape`. It instantiates an array of objects derived from `Shape` and calls the static methods of the `Shape` class, which wraps return `Shape` property values. The runtime retrieves values from the overridden properties of the derived types. The example also casts each `Shape` object in the array to its derived type and, if the cast succeeds, retrieves properties of that particular subclass of `Shape`.


```
using System;

public class Example
{
    public static void Main()
    {
        Shape[] shapes = { new Rectangle(10, 12), new Square(5),
                           new Circle(3) };
        foreach (var shape in shapes) {
            Console.WriteLine($"{shape}: area, {Shape.GetArea(shape)}; " +
                              $"perimeter, {Shape.GetPerimeter(shape)}");
            var rect = shape as Rectangle;
            if (rect != null) {
                Console.WriteLine($"    Is Square: {rect.IsSquare()}, Diagonal: {rect.Diagonal}");
                continue;
            }
            var sq = shape as Square;
            if (sq != null) {
                Console.WriteLine($"    Diagonal: {sq.Diagonal}");
                continue;
            }
        }
    }
}

// The example displays the following output:
//      Rectangle: area, 120; perimeter, 44
//      Is Square: False, Diagonal: 15.62
//      Square: area, 25; perimeter, 20
//      Diagonal: 7.07
//      Circle: area, 28.27; perimeter, 18.85
```

How to safely cast by using pattern matching and the is and as operators

12/28/2021 • 4 minutes to read • [Edit Online](#)

Because objects are polymorphic, it's possible for a variable of a base class type to hold a derived [type](#). To access the derived type's instance members, it's necessary to [cast](#) the value back to the derived type. However, a cast creates the risk of throwing an [InvalidCastException](#). C# provides [pattern matching](#) statements that perform a cast conditionally only when it will succeed. C# also provides the [is](#) and [as](#) operators to test if a value is of a certain type.

The following example shows how to use the pattern matching `is` statement:

```

class Animal
{
    public void Eat() { Console.WriteLine("Eating."); }
    public override string ToString()
    {
        return "I am an animal.";
    }
}
class Mammal : Animal { }
class Giraffe : Mammal { }

class SuperNova { }

class Program
{
    static void Main(string[] args)
    {
        var g = new Giraffe();
        var a = new Animal();
        FeedMammals(g);
        FeedMammals(a);
        // Output:
        // Eating.
        // Animal is not a Mammal

        SuperNova sn = new SuperNova();
        TestForMammals(g);
        TestForMammals(sn);
        // Output:
        // I am an animal.
        // SuperNova is not a Mammal
    }

    static void FeedMammals(Animal a)
    {
        if (a is Mammal m)
        {
            m.Eat();
        }
        else
        {
            // variable 'm' is not in scope here, and can't be used.
            Console.WriteLine($"{a.GetType().Name} is not a Mammal");
        }
    }

    static void TestForMammals(object o)
    {
        // You also can use the as operator and test for null
        // before referencing the variable.
        var m = o as Mammal;
        if (m != null)
        {
            Console.WriteLine(m.ToString());
        }
        else
        {
            Console.WriteLine($"{o.GetType().Name} is not a Mammal");
        }
    }
}

```

The preceding sample demonstrates a few features of pattern matching syntax. The `if (a is Mammal m)` statement combines the test with an initialization assignment. The assignment occurs only when the test succeeds. The variable `m` is only in scope in the embedded `if` statement where it has been assigned. You can't

access `m` later in the same method. The preceding example also shows how to use the `as` operator to convert an object to a specified type.

You can also use the same syntax for testing if a `nullable value type` has a value, as shown in the following example:

```

class Program
{
    static void Main(string[] args)
    {
        int i = 5;
        PatternMatchingNullable(i);

        int? j = null;
        PatternMatchingNullable(j);

        double d = 9.78654;
        PatternMatchingNullable(d);

        PatternMatchingSwitch(i);
        PatternMatchingSwitch(j);
        PatternMatchingSwitch(d);
    }

    static void PatternMatchingNullable(System.ValueType val)
    {
        if (val is int j) // Nullable types are not allowed in patterns
        {
            Console.WriteLine(j);
        }
        else if (val is null) // If val is a nullable type with no value, this expression is true
        {
            Console.WriteLine("val is a nullable type with the null value");
        }
        else
        {
            Console.WriteLine("Could not convert " + val.ToString());
        }
    }

    static void PatternMatchingSwitch(System.ValueType val)
    {
        switch (val)
        {
            case int number:
                Console.WriteLine(number);
                break;
            case long number:
                Console.WriteLine(number);
                break;
            case decimal number:
                Console.WriteLine(number);
                break;
            case float number:
                Console.WriteLine(number);
                break;
            case double number:
                Console.WriteLine(number);
                break;
            case null:
                Console.WriteLine("val is a nullable type with the null value");
                break;
            default:
                Console.WriteLine("Could not convert " + val.ToString());
                break;
        }
    }
}

```

The preceding sample demonstrates other features of pattern matching to use with conversions. You can test a variable for the null pattern by checking specifically for the `null` value. When the runtime value of the variable is `null`, an `is` statement checking for a type always returns `false`. The pattern matching `is` statement

doesn't allow a nullable value type, such as `int?` or `Nullable<int>`, but you can test for any other value type. The `is` patterns from the preceding example aren't limited to the nullable value types. You can also use those patterns to test if a variable of a reference type has a value or it's `null`.

The preceding sample also shows how you use the type pattern in a `switch` statement where the variable may be one of many different types.

If you want to test if a variable is a given type, but not assign it to a new variable, you can use the `is` and `as` operators for reference types and nullable value types. The following code shows how to use the `is` and `as` statements that were part of the C# language before pattern matching was introduced to test if a variable is of a given type:

```
class Animal
{
    public void Eat() { Console.WriteLine("Eating."); }
    public override string ToString()
    {
        return "I am an animal.";
    }
}
class Mammal : Animal { }
class Giraffe : Mammal { }

class SuperNova { }

class Program
{
    static void Main(string[] args)
    {
        // Use the is operator to verify the type.
        // before performing a cast.
        Giraffe g = new Giraffe();
        UseIsOperator(g);

        // Use the as operator and test for null
        // before referencing the variable.
        UseAsOperator(g);

        // Use the as operator to test
        // an incompatible type.
        SuperNova sn = new SuperNova();
        UseAsOperator(sn);

        // Use the as operator with a value type.
        // Note the implicit conversion to int? in
        // the method body.
        int i = 5;
        UseAsWithNullable(i);

        double d = 9.78654;
        UseAsWithNullable(d);
    }

    static void UseIsOperator(Animal a)
    {
        if (a is Mammal)
        {
            Mammal m = (Mammal)a;
            m.Eat();
        }
    }

    static void UsePatternMatchingIs(Animal a)
    {
        if (a is Mammal m)
        {
            m.Eat();
        }
    }
}
```

```

        {
            m.Eat();
        }
    }

    static void UseAsOperator(object o)
    {
        Mammal m = o as Mammal;
        if (m != null)
        {
            Console.WriteLine(m.ToString());
        }
        else
        {
            Console.WriteLine($"{o.GetType().Name} is not a Mammal");
        }
    }

    static void UseAsWithNullable(System.ValueType val)
    {
        int? j = val as int?;
        if (j != null)
        {
            Console.WriteLine(j);
        }
        else
        {
            Console.WriteLine("Could not convert " + val.ToString());
        }
    }
}

```

As you can see by comparing this code with the pattern matching code, the pattern matching syntax provides more robust features by combining the test and the assignment in a single statement. Use the pattern matching syntax whenever possible.

Tutorial: Use pattern matching to build type-driven and data-driven algorithms

12/28/2021 • 15 minutes to read • [Edit Online](#)

C# 7 introduced basic pattern matching features. Those features are extended in C# 8 through C# 10 with new expressions and patterns. You can write functionality that behaves as though you extended types that may be in other libraries. Another use for patterns is to create functionality your application requires that isn't a fundamental feature of the type being extended.

In this tutorial, you'll learn how to:

- Recognize situations where pattern matching should be used.
- Use pattern matching expressions to implement behavior based on types and property values.
- Combine pattern matching with other techniques to create complete algorithms.

Prerequisites

You'll need to set up your machine to run .NET 6, which includes the C# 10 compiler. The C# 10 compiler is available starting with [Visual Studio 2022](#) or [.NET 6 SDK](#).

This tutorial assumes you're familiar with C# and .NET, including either Visual Studio or the .NET CLI.

Scenarios for pattern matching

Modern development often includes integrating data from multiple sources and presenting information and insights from that data in a single cohesive application. You and your team won't have control or access for all the types that represent the incoming data.

The classic object-oriented design would call for creating types in your application that represent each data type from those multiple data sources. Then, your application would work with those new types, build inheritance hierarchies, create virtual methods, and implement abstractions. Those techniques work, and sometimes they are the best tools. Other times you can write less code. You can write more clear code using techniques that separate the data from the operations that manipulate that data.

In this tutorial, you'll create and explore an application that takes incoming data from several external sources for a single scenario. You'll see how **pattern matching** provides an efficient way to consume and process that data in ways that weren't part of the original system.

Consider a major metropolitan area that is using tolls and peak time pricing to manage traffic. You write an application that calculates tolls for a vehicle based on its type. Later enhancements incorporate pricing based on the number of occupants in the vehicle. Further enhancements add pricing based on the time and the day of the week.

From that brief description, you may have quickly sketched out an object hierarchy to model this system. However, your data is coming from multiple sources like other vehicle registration management systems. These systems provide different classes to model that data and you don't have a single object model you can use. In this tutorial, you'll use these simplified classes to model for the vehicle data from these external systems, as shown in the following code:


```

namespace ConsumerVehicleRegistration
{
    public class Car
    {
        public int Passengers { get; set; }
    }
}

namespace CommercialRegistration
{
    public class DeliveryTruck
    {
        public int GrossWeightClass { get; set; }
    }
}

namespace LiveryRegistration
{
    public class Taxi
    {
        public int Fares { get; set; }
    }

    public class Bus
    {
        public int Capacity { get; set; }
        public int Riders { get; set; }
    }
}

```

You can download the starter code from the [dotnet/samples](#) GitHub repository. You can see that the vehicle classes are from different systems, and are in different namespaces. No common base class, other than `System.Object` can be leveraged.

Pattern matching designs

The scenario used in this tutorial highlights the kinds of problems that pattern matching is well suited to solve:

- The objects you need to work with aren't in an object hierarchy that matches your goals. You may be working with classes that are part of unrelated systems.
- The functionality you're adding isn't part of the core abstraction for these classes. The toll paid by a vehicle *changes* for different types of vehicles, but the toll isn't a core function of the vehicle.

When the *shape* of the data and the *operations* on that data are not described together, the pattern matching features in C# make it easier to work with.

Implement the basic toll calculations

The most basic toll calculation relies only on the vehicle type:

- A `Car` is \$2.00.
- A `Taxi` is \$3.50.
- A `Bus` is \$5.00.
- A `DeliveryTruck` is \$10.00

Create a new `TollCalculator` class, and implement pattern matching on the vehicle type to get the toll amount. The following code shows the initial implementation of the `TollCalculator`.

```

using System;
using CommercialRegistration;
using ConsumerVehicleRegistration;
using LiveryRegistration;

namespace toll_calculator
{
    public class TollCalculator
    {
        public decimal CalculateToll(object vehicle) =>
            vehicle switch
            {
                Car c          => 2.00m,
                Taxi t          => 3.50m,
                Bus b           => 5.00m,
                DeliveryTruck t => 10.00m,
                { }             => throw new ArgumentException(message: "Not a known vehicle type", paramName:
nameof(vehicle)),
                null           => throw new ArgumentNullException(nameof(vehicle))
            };
    }
}

```

The preceding code uses a `switch expression` (not the same as a `switch statement`) that tests the `declaration pattern`. A `switch expression` begins with the variable, `vehicle` in the preceding code, followed by the `switch` keyword. Next comes all the `switch arms` inside curly braces. The `switch` expression makes other refinements to the syntax that surrounds the `switch` statement. The `case` keyword is omitted, and the result of each arm is an expression. The last two arms show a new language feature. The `{ }` case matches any non-null object that didn't match an earlier arm. This arm catches any incorrect types passed to this method. The `{ }` case must follow the cases for each vehicle type. If the order were reversed, the `{ }` case would take precedence. Finally, the `null` `constant pattern` detects when `null` is passed to this method. The `null` pattern can be last because the other patterns match only a non-null object of the correct type.

You can test this code using the following code in `Program.cs` :

```

using System;
using CommercialRegistration;
using ConsumerVehicleRegistration;
using LiveryRegistration;

namespace toll_calculator
{
    class Program
    {
        static void Main(string[] args)
        {
            var tollCalc = new TollCalculator();

            var car = new Car();
            var taxi = new Taxi();
            var bus = new Bus();
            var truck = new DeliveryTruck();

            Console.WriteLine($"The toll for a car is {tollCalc.CalculateToll(car)}");
            Console.WriteLine($"The toll for a taxi is {tollCalc.CalculateToll(taxi)}");
            Console.WriteLine($"The toll for a bus is {tollCalc.CalculateToll(bus)}");
            Console.WriteLine($"The toll for a truck is {tollCalc.CalculateToll(truck)}");

            try
            {
                tollCalc.CalculateToll("this will fail");
            }
            catch (ArgumentException e)
            {
                Console.WriteLine("Caught an argument exception when using the wrong type");
            }
            try
            {
                tollCalc.CalculateToll(null!);
            }
            catch (ArgumentNullException e)
            {
                Console.WriteLine("Caught an argument exception when using null");
            }
        }
    }
}

```

That code is included in the starter project, but is commented out. Remove the comments, and you can test what you've written.

You're starting to see how patterns can help you create algorithms where the code and the data are separate. The `switch` expression tests the type and produces different values based on the results. That's only the beginning.

Add occupancy pricing

The toll authority wants to encourage vehicles to travel at maximum capacity. They've decided to charge more when vehicles have fewer passengers, and encourage full vehicles by offering lower pricing:

- Cars and taxis with no passengers pay an extra \$0.50.
- Cars and taxis with two passengers get a \$0.50 discount.
- Cars and taxis with three or more passengers get a \$1.00 discount.
- Buses that are less than 50% full pay an extra \$2.00.
- Buses that are more than 90% full get a \$1.00 discount.

These rules can be implemented using a [property pattern](#) in the same switch expression. A property pattern

compares a property value to a constant value. The property pattern examines properties of the object once the type has been determined. The single case for a `Car` expands to four different cases:

```
vehicle switch
{
    Car {Passengers: 0} => 2.00m + 0.50m,
    Car {Passengers: 1} => 2.0m,
    Car {Passengers: 2} => 2.0m - 0.50m,
    Car                    => 2.00m - 1.0m,

    // ...
};
```

The first three cases test the type as a `Car`, then check the value of the `Passengers` property. If both match, that expression is evaluated and returned.

You would also expand the cases for taxis in a similar manner:

```
vehicle switch
{
    // ...

    Taxi {Fares: 0}  => 3.50m + 1.00m,
    Taxi {Fares: 1}  => 3.50m,
    Taxi {Fares: 2}  => 3.50m - 0.50m,
    Taxi             => 3.50m - 1.00m,

    // ...
};
```

Next, implement the occupancy rules by expanding the cases for buses, as shown in the following example:

```
vehicle switch
{
    // ...

    Bus b when ((double)b.Riders / (double)b.Capacity) < 0.50 => 5.00m + 2.00m,
    Bus b when ((double)b.Riders / (double)b.Capacity) > 0.90 => 5.00m - 1.00m,
    Bus => 5.00m,

    // ...
};
```

The toll authority isn't concerned with the number of passengers in the delivery trucks. Instead, they adjust the toll amount based on the weight class of the trucks as follows:

- Trucks over 5000 lbs are charged an extra \$5.00.
- Light trucks under 3000 lbs are given a \$2.00 discount.

That rule is implemented with the following code:

```
vehicle switch
{
    // ...

    DeliveryTruck t when (t.GrossWeightClass > 5000) => 10.00m + 5.00m,
    DeliveryTruck t when (t.GrossWeightClass < 3000) => 10.00m - 2.00m,
    DeliveryTruck => 10.00m,

};
```

The preceding code shows the `when` clause of a switch arm. You use the `when` clause to test conditions other than equality on a property. When you've finished, you'll have a method that looks much like the following code:

```
vehicle switch
{
    Car {Passengers: 0}      => 2.00m + 0.50m,
    Car {Passengers: 1}      => 2.0m,
    Car {Passengers: 2}      => 2.0m - 0.50m,
    Car                      => 2.00m - 1.0m,

    Taxi {Fares: 0}          => 3.50m + 1.00m,
    Taxi {Fares: 1}          => 3.50m,
    Taxi {Fares: 2}          => 3.50m - 0.50m,
    Taxi                    => 3.50m - 1.00m,

    Bus b when ((double)b.Riders / (double)b.Capacity) < 0.50 => 5.00m + 2.00m,
    Bus b when ((double)b.Riders / (double)b.Capacity) > 0.90 => 5.00m - 1.00m,
    Bus                     => 5.00m,

    DeliveryTruck t when (t.GrossWeightClass > 5000) => 10.00m + 5.00m,
    DeliveryTruck t when (t.GrossWeightClass < 3000) => 10.00m - 2.00m,
    DeliveryTruck           => 10.00m,

    { }                     => throw new ArgumentException(message: "Not a known vehicle type", paramName: nameof(vehicle)),
    null                    => throw new ArgumentNullException(nameof(vehicle))
};
```

Many of these switch arms are examples of **recursive patterns**. For example, `Car { Passengers: 1}` shows a constant pattern inside a property pattern.

You can make this code less repetitive by using nested switches. The `Car` and `Taxi` both have four different arms in the preceding examples. In both cases, you can create a declaration pattern that feeds into a constant pattern. This technique is shown in the following code:

```

public decimal CalculateToll(object vehicle) =>
    vehicle switch
    {
        Car c => c.Passengers switch
        {
            0 => 2.00m + 0.5m,
            1 => 2.0m,
            2 => 2.0m - 0.5m,
            _ => 2.00m - 1.0m
        },

        Taxi t => t.Fares switch
        {
            0 => 3.50m + 1.00m,
            1 => 3.50m,
            2 => 3.50m - 0.50m,
            _ => 3.50m - 1.00m
        },

        Bus b when ((double)b.Riders / (double)b.Capacity) < 0.50 => 5.00m + 2.00m,
        Bus b when ((double)b.Riders / (double)b.Capacity) > 0.90 => 5.00m - 1.00m,
        Bus b => 5.00m,

        DeliveryTruck t when (t.GrossWeightClass > 5000) => 10.00m + 5.00m,
        DeliveryTruck t when (t.GrossWeightClass < 3000) => 10.00m - 2.00m,
        DeliveryTruck t => 10.00m,

        { } => throw new ArgumentException(message: "Not a known vehicle type", paramName:
nameof(vehicle)),
        null => throw new ArgumentNullException(nameof(vehicle))
    };

```

In the preceding sample, using a recursive expression means you don't repeat the `Car` and `Taxi` arms containing child arms that test the property value. This technique isn't used for the `Bus` and `DeliveryTruck` arms because those arms are testing ranges for the property, not discrete values.

Add peak pricing

For the final feature, the toll authority wants to add time sensitive peak pricing. During the morning and evening rush hours, the tolls are doubled. That rule only affects traffic in one direction: inbound to the city in the morning, and outbound in the evening rush hour. During other times during the workday, tolls increase by 50%. Late night and early morning, tolls are reduced by 25%. During the weekend, it's the normal rate, regardless of the time. You could use a series of `if` and `else` statements to express this using the following code:

```

public decimal PeakTimePremiumIfElse(DateTime timeOfToll, bool inbound)
{
    if ((timeOfToll.DayOfWeek == DayOfWeek.Saturday) ||
        (timeOfToll.DayOfWeek == DayOfWeek.Sunday))
    {
        return 1.0m;
    }
    else
    {
        int hour = timeOfToll.Hour;
        if (hour < 6)
        {
            return 0.75m;
        }
        else if (hour < 10)
        {
            if (inbound)
            {
                return 2.0m;
            }
            else
            {
                return 1.0m;
            }
        }
        else if (hour < 16)
        {
            return 1.5m;
        }
        else if (hour < 20)
        {
            if (inbound)
            {
                return 1.0m;
            }
            else
            {
                return 2.0m;
            }
        }
        else // Overnight
        {
            return 0.75m;
        }
    }
}

```

The preceding code does work correctly, but isn't readable. You have to chain through all the input cases and the nested `if` statements to reason about the code. Instead, you'll use pattern matching for this feature, but you'll integrate it with other techniques. You could build a single pattern match expression that would account for all the combinations of direction, day of the week, and time. The result would be a complicated expression. It would be hard to read and difficult to understand. That makes it hard to ensure correctness. Instead, combine those methods to build a tuple of values that concisely describes all those states. Then use pattern matching to calculate a multiplier for the toll. The tuple contains three discrete conditions:

- The day is either a weekday or a weekend.
- The band of time when the toll is collected.
- The direction is into the city or out of the city

The following table shows the combinations of input values and the peak pricing multiplier:

DAY	TIME	DIRECTION	PREMIUM
Weekday	morning rush	inbound	x 2.00
Weekday	morning rush	outbound	x 1.00
Weekday	daytime	inbound	x 1.50
Weekday	daytime	outbound	x 1.50
Weekday	evening rush	inbound	x 1.00
Weekday	evening rush	outbound	x 2.00
Weekday	overnight	inbound	x 0.75
Weekday	overnight	outbound	x 0.75
Weekend	morning rush	inbound	x 1.00
Weekend	morning rush	outbound	x 1.00
Weekend	daytime	inbound	x 1.00
Weekend	daytime	outbound	x 1.00
Weekend	evening rush	inbound	x 1.00
Weekend	evening rush	outbound	x 1.00
Weekend	overnight	inbound	x 1.00
Weekend	overnight	outbound	x 1.00

There are 16 different combinations of the three variables. By combining some of the conditions, you'll simplify the final switch expression.

The system that collects the tolls uses a [DateTime](#) structure for the time when the toll was collected. Build member methods that create the variables from the preceding table. The following function uses a pattern matching switch expression to express whether a [DateTime](#) represents a weekend or a weekday:

```
private static bool IsWeekDay(DateTime timeOfToll) =>
    timeOfToll.DayOfWeek switch
    {
        DayOfWeek.Monday    => true,
        DayOfWeek.Tuesday   => true,
        DayOfWeek.Wednesday => true,
        DayOfWeek.Thursday  => true,
        DayOfWeek.Friday    => true,
        DayOfWeek.Saturday  => false,
        DayOfWeek.Sunday    => false
    };
```

That method is correct, but it's repetitious. You can simplify it, as shown in the following code:


```
private static bool IsWeekDay(DateTime timeOfToll) =>
    timeOfToll.DayOfWeek switch
    {
        DayOfWeek.Saturday => false,
        DayOfWeek.Sunday => false,
        _ => true
    };
```

Next, add a similar function to categorize the time into the blocks:

```
private enum TimeBand
{
    MorningRush,
    Daytime,
    EveningRush,
    Overnight
}

private static TimeBand GetTimeBand(DateTime timeOfToll) =>
    timeOfToll.Hour switch
    {
        < 6 or > 19 => TimeBand.Overnight,
        < 10 => TimeBand.MorningRush,
        < 16 => TimeBand.Daytime,
        _ => TimeBand.EveningRush,
    };
```

You add a private `enum` to convert each range of time to a discrete value. Then, the `GetTimeBand` method uses [relational patterns](#), and [conjunctive or patterns](#), both added in C# 9.0. A relational pattern lets you test a numeric value using `<`, `>`, `<=`, or `>=`. The `or` pattern tests if an expression matches one or more patterns. You can also use an `and` pattern to ensure that an expression matches two distinct patterns, and a `not` pattern to test that an expression doesn't match a pattern.

After you create those methods, you can use another `switch` expression with the **tuple pattern** to calculate the pricing premium. You could build a `switch` expression with all 16 arms:

```
public decimal PeakTimePremiumFull(DateTime timeOfToll, bool inbound) =>
    (IsWeekDay(timeOfToll), GetTimeBand(timeOfToll), inbound) switch
    {
        (true, TimeBand.MorningRush, true) => 2.00m,
        (true, TimeBand.MorningRush, false) => 1.00m,
        (true, TimeBand.Daytime, true) => 1.50m,
        (true, TimeBand.Daytime, false) => 1.50m,
        (true, TimeBand.EveningRush, true) => 1.00m,
        (true, TimeBand.EveningRush, false) => 2.00m,
        (true, TimeBand.Overnight, true) => 0.75m,
        (true, TimeBand.Overnight, false) => 0.75m,
        (false, TimeBand.MorningRush, true) => 1.00m,
        (false, TimeBand.MorningRush, false) => 1.00m,
        (false, TimeBand.Daytime, true) => 1.00m,
        (false, TimeBand.Daytime, false) => 1.00m,
        (false, TimeBand.EveningRush, true) => 1.00m,
        (false, TimeBand.EveningRush, false) => 1.00m,
        (false, TimeBand.Overnight, true) => 1.00m,
        (false, TimeBand.Overnight, false) => 1.00m,
    };
```

The above code works, but it can be simplified. All eight combinations for the weekend have the same toll. You can replace all eight with the following line:

```
(false, _, _) => 1.0m,
```

Both inbound and outbound traffic have the same multiplier during the weekday daytime and overnight hours. Those four switch arms can be replaced with the following two lines:

```
(true, TimeBand.Overnight, _) => 0.75m,  
(true, TimeBand.Daytime, _) => 1.5m,
```

The code should look like the following code after those two changes:

```
public decimal PeakTimePremium(DateTime timeOfToll, bool inbound) =>  
    (IsWeekDay(timeOfToll), GetTimeBand(timeOfToll), inbound) switch  
    {  
        (true, TimeBand.MorningRush, true) => 2.00m,  
        (true, TimeBand.MorningRush, false) => 1.00m,  
        (true, TimeBand.Daytime, _) => 1.50m,  
        (true, TimeBand.EveningRush, true) => 1.00m,  
        (true, TimeBand.EveningRush, false) => 2.00m,  
        (true, TimeBand.Overnight, _) => 0.75m,  
        (false, _, _) => 1.00m,  
    };
```

Finally, you can remove the two rush hour times that pay the regular price. Once you remove those arms, you can replace the `false` with a discard (`_`) in the final switch arm. You'll have the following finished method:

```
public decimal PeakTimePremium(DateTime timeOfToll, bool inbound) =>  
    (IsWeekDay(timeOfToll), GetTimeBand(timeOfToll), inbound) switch  
    {  
        (true, TimeBand.Overnight, _) => 0.75m,  
        (true, TimeBand.Daytime, _) => 1.5m,  
        (true, TimeBand.MorningRush, true) => 2.0m,  
        (true, TimeBand.EveningRush, false) => 2.0m,  
        _ => 1.0m,  
    };
```

This example highlights one of the advantages of pattern matching: the pattern branches are evaluated in order. If you rearrange them so that an earlier branch handles one of your later cases, the compiler warns you about the unreachable code. Those language rules made it easier to do the preceding simplifications with confidence that the code didn't change.

Pattern matching makes some types of code more readable and offers an alternative to object-oriented techniques when you can't add code to your classes. The cloud is causing data and functionality to live apart. The *shape* of the data and the *operations* on it aren't necessarily described together. In this tutorial, you consumed existing data in entirely different ways from its original function. Pattern matching gave you the ability to write functionality that overrode those types, even though you couldn't extend them.

Next steps

You can download the finished code from the [dotnet/samples](#) GitHub repository. Explore patterns on your own and add this technique into your regular coding activities. Learning these techniques gives you another way to approach problems and create new functionality.

See also

- [Patterns](#)

- `switch` expression

How to handle an exception using try/catch

12/28/2021 • 2 minutes to read • [Edit Online](#)

The purpose of a [try-catch](#) block is to catch and handle an exception generated by working code. Some exceptions can be handled in a `catch` block and the problem solved without the exception being rethrown; however, more often the only thing that you can do is make sure that the appropriate exception is thrown.

Example

In this example, [IndexOutOfRangeException](#) isn't the most appropriate exception:

[ArgumentOutOfRangeException](#) makes more sense for the method because the error is caused by the `index` argument passed in by the caller.

```
static int GetInt(int[] array, int index)
{
    try
    {
        return array[index];
    }
    catch (IndexOutOfRangeException e) // CS0168
    {
        Console.WriteLine(e.Message);
        // Set IndexOutOfRangeException to the new exception's InnerException.
        throw new ArgumentOutOfRangeException("index parameter is out of range.", e);
    }
}
```

Comments

The code that causes an exception is enclosed in the `try` block. A `catch` statement is added immediately after to handle `IndexOutOfRangeException`, if it occurs. The `catch` block handles the `IndexOutOfRangeException` and throws the more appropriate `ArgumentOutOfRangeException` exception instead. In order to provide the caller with as much information as possible, consider specifying the original exception as the [InnerException](#) of the new exception. Because the [InnerException](#) property is [read-only](#), you must assign it in the constructor of the new exception.

How to execute cleanup code using finally

12/28/2021 • 2 minutes to read • [Edit Online](#)

The purpose of a `finally` statement is to ensure that the necessary cleanup of objects, usually objects that are holding external resources, occurs immediately, even if an exception is thrown. One example of such cleanup is calling `Close` on a `FileStream` immediately after use instead of waiting for the object to be garbage collected by the common language runtime, as follows:

```
static void CodeWithoutCleanup()
{
    FileStream? file = null;
    FileInfo fileInfo = new FileInfo("./file.txt");

    file = fileInfo.OpenWrite();
    file.WriteByte(0xF);

    file.Close();
}
```

Example

To turn the previous code into a `try-catch-finally` statement, the cleanup code is separated from the working code, as follows.

```
static void CodeWithCleanup()
{
    FileStream? file = null;
    FileInfo? fileInfo = null;

    try
    {
        fileInfo = new FileInfo("./file.txt");

        file = fileInfo.OpenWrite();
        file.WriteByte(0xF);
    }
    catch (UnauthorizedAccessException e)
    {
        Console.WriteLine(e.Message);
    }
    finally
    {
        file?.Close();
    }
}
```

Because an exception can occur at any time within the `try` block before the `OpenWrite()` call, or the `OpenWrite()` call itself could fail, we aren't guaranteed that the file is open when we try to close it. The `finally` block adds a check to make sure that the `FileStream` object isn't `null` before you call the `Close` method. Without the `null` check, the `finally` block could throw its own `NullReferenceException`, but throwing exceptions in `finally` blocks should be avoided if it's possible.

A database connection is another good candidate for being closed in a `finally` block. Because the number of connections allowed to a database server is sometimes limited, you should close database connections as

quickly as possible. If an exception is thrown before you can close your connection, using the `finally` block is better than waiting for garbage collection.

See also

- [using Statement](#)
- [try-catch](#)
- [try-finally](#)
- [try-catch-finally](#)

What's new in C# 10

12/28/2021 • 7 minutes to read • [Edit Online](#)

C# 10 adds the following features and enhancements to the C# language:

- [Record structs](#)
- [Improvements of structure types](#)
- [Interpolated string handlers](#)
- [global using directives](#)
- [File-scoped namespace declaration](#)
- [Extended property patterns](#)
- [Improvements on lambda expressions](#)
- [Allow `const` interpolated strings](#)
- [Record types can seal `ToString\(\)`](#)
- [Improved definite assignment](#)
- [Allow both assignment and declaration in the same deconstruction](#)
- [Allow `AsyncMethodBuilder` attribute on methods](#)
- [CallerArgumentExpression attribute](#)
- [Enhanced `#line` pragma](#)

Additional features are available in *preview* mode. You're encouraged to try these features and provide feedback on them. They may change before their final release. In order to use these features, you must [set `<LangVersion>` to `Preview`](#) in your project. Read about [Generic attributes](#) later in this article.

C# 10 is supported on .NET 6. For more information, see [C# language versioning](#).

You can download the latest .NET 6 SDK from the [.NET downloads page](#). You can also download [Visual Studio 2022](#), which includes the .NET 6 SDK.

Record structs

You can declare value type records using the [record struct](#) or [readonly record struct](#) declarations. You can now clarify that a [record](#) is a reference type with the [record class](#) declaration.

Improvements of structure types

C# 10 introduces the following improvements related to structure types:

- You can declare an instance parameterless constructor in a structure type and initialize an instance field or property at its declaration. For more information, see the [Parameterless constructors and field initializers](#) section of the [Structure types](#) article.
- A left-hand operand of the [with expression](#) can be of any structure type or an anonymous (reference) type.

Interpolated string handler

You can create a type that builds the resulting string from an [interpolated string expression](#). The .NET libraries use this feature in many APIs. You can build one by [following this tutorial](#).

Global using directives

You can add the `global` modifier to any [using directive](#) to instruct the compiler that the directive applies to all source files in the compilation. This is typically all source files in a project.

File-scoped namespace declaration

You can use a new form of the `namespace` [declaration](#) to declare that all declarations that follow are members of the declared namespace:

```
namespace MyNamespace;
```

This new syntax saves both horizontal and vertical space for `namespace` declarations.

Extended property patterns

Beginning with C# 10, you can reference nested properties or fields within a property pattern. For example, a pattern of the form

```
{ Prop1.Prop2: pattern }
```

is valid in C# 10 and later and equivalent to

```
{ Prop1: { Prop2: pattern } }
```

valid in C# 8.0 and later.

For more information, see the [Extended property patterns](#) feature proposal note. For more information about a property pattern, see the [Property pattern](#) section of the [Patterns](#) article.

Lambda expression improvements

C# 10 includes many improvements to how lambda expressions are handled:

- Lambda expressions may have a [natural type](#), where the compiler can infer a delegate type from the lambda expression or method group.
- Lambda expressions may declare a [return type](#) when the compiler can't infer it.
- [Attributes](#) can be applied to lambda expressions.

These features make lambda expressions more similar to methods and local functions. They make it easier to use lambda expressions without declaring a variable of a delegate type, and they work more seamlessly with the new ASP.NET Core Minimal APIs.

Constant interpolated strings

In C# 10, `const` strings may be initialized using [string interpolation](#) if all the placeholders are themselves constant strings. String interpolation can create more readable constant strings as you build constant strings used in your application. The placeholder expressions can't be numeric constants because those constants are converted to strings at run time. The current culture may affect their string representation. Learn more in the language reference on [const expressions](#).

Record types can seal ToString

In C# 10, you can add the `sealed` modifier when you override `ToString` in a record type. Sealing the `ToString`

method prevents the compiler from synthesizing a `Tostring` method for any derived record types. A `sealed` `Tostring` ensures all derived record types use the `Tostring` method defined in a common base record type. You can learn more about this feature in the article on [records](#).

Assignment and declaration in same deconstruction

This change removes a restriction from earlier versions of C#. Previously, a deconstruction could assign all values to existing variables, or initialize newly declared variables:

```
// Initialization:
(int x, int y) = point;

// assignment:
int x1 = 0;
int y1 = 0;
(x1, y1) = point;
```

C# 10 removes this restriction:

```
int x = 0;
(x, int y) = point;
```

Improved definite assignment

Prior to C# 10, there were many scenarios where definite assignment and null-state analysis produced warnings that were false positives. These generally involved comparisons to boolean constants, accessing a variable only in the `true` or `false` statements in an `if` statement, and null coalescing expressions. These examples generated warnings in previous versions of C#, but don't in C# 10:

```
string representation = "N/A";
if ((c != null && c.GetDependentValue(out object obj)) == true)
{
    representation = obj.ToString(); // undesired error
}

// Or, using ?.
if (c?.GetDependentValue(out object obj) == true)
{
    representation = obj.ToString(); // undesired error
}

// Or, using ??
if (c?.GetDependentValue(out object obj) ?? false)
{
    representation = obj.ToString(); // undesired error
}
```

The main impact of this improvement is that the warnings for definite assignment and null-state analysis are more accurate.

Allow AsyncMethodBuilder attribute on methods

In C# 10 and later, you can specify a different async method builder for a single method, in addition to specifying the method builder type for all methods that return a given task-like type. A custom async method builder enables advanced performance tuning scenarios where a given method may benefit from a custom builder.

To learn more, see the section on [AsyncMethodBuilder](#) in the article on attributes read by the compiler.

CallerArgumentExpression attribute diagnostics

You can use the [System.Runtime.CompilerServices.CallerArgumentExpressionAttribute](#) to specify a parameter that the compiler replaces with the text representation of another argument. This feature enables libraries to create more specific diagnostics. The following code tests a condition. If the condition is false, the exception message contains the text representation of the argument passed to `condition`:

```
public static void Validate(bool condition, [CallerArgumentExpression("condition")] string? message=null)
{
    if (!condition)
    {
        throw new InvalidOperationException($"Argument failed validation: <{message}>");
    }
}
```

You can learn more about this feature in the article on [Caller information attributes](#) in the language reference section.

Enhanced #line pragma

C# 10 supports a new format for the `#line` pragma. You likely won't use the new format, but you'll see its effects. The enhancements enable more fine-grained output in domain-specific languages (DSLs) like Razor. The Razor engine uses these enhancements to improve the debugging experience. You'll find debuggers can highlight your Razor source more accurately. To learn more about the new syntax, see the article on [Preprocessor directives](#) in the language reference. You can also read the [feature specification](#) for Razor based examples.

Generic attributes

IMPORTANT

Generic attributes is a preview feature. You must [set](#) `<LangVersion>` to `Preview` to enable this feature. This feature may change before its final release.

You can declare a [generic class](#) whose base class is [System.Attribute](#). This provides a more convenient syntax for attributes that require a [System.Type](#) parameter. Previously, you'd need to create an attribute that takes a `Type` as its constructor parameter:

```
public class TypeAttribute : Attribute
{
    public TypeAttribute(Type t) => ParamType = t;

    public Type ParamType { get; }
}
```

And to apply the attribute, you use the `typeof` operator:

```
[TypeAttribute(typeof(string))]
public string Method() => default;
```

Using this new feature, you can create a generic attribute instead:

```
public class GenericAttribute<T> : Attribute { }
```

Then, specify the type parameter to use the attribute:

```
[GenericAttribute<string>()]  
public string Method() => default;
```

You can apply a fully closed constructed generic attribute. In other words, all type parameters must be specified. For example, the following is not allowed:

```
public class GenericType<T>  
{  
    [GenericAttribute<T>()] // Not allowed! generic attributes must be fully closed types.  
    public string Method() => default;  
}
```

The type arguments must satisfy the same restrictions as the `typeof` operator. Types that require metadata annotations aren't allowed. Examples include the following:

- `dynamic`
- `nint`, `nuint`
- `string?` (or any nullable reference type)
- `(int X, int Y)` (or any other tuple types using C# tuple syntax).

These types aren't directly represented in metadata. They include annotations that describe the type. In all cases, you can use the underlying type instead:

- `object` for `dynamic`.
- `IntPtr` instead of `nint` or `nuint`.
- `string` instead of `string?`.
- `ValueTuple<int, int>` instead of `(int X, int Y)`.

What's new in C# 9.0

12/28/2021 • 17 minutes to read • [Edit Online](#)

C# 9.0 adds the following features and enhancements to the C# language:

- [Records](#)
- [Init only setters](#)
- [Top-level statements](#)
- [Pattern matching enhancements](#)
- [Performance and interop](#)
 - [Native sized integers](#)
 - [Function pointers](#)
 - [Suppress emitting localsinit flag](#)
- [Fit and finish features](#)
 - [Target-typed `new` expressions](#)
 - [static anonymous functions](#)
 - [Target-typed conditional expressions](#)
 - [Covariant return types](#)
 - [Extension `GetEnumerator` support for `foreach` loops](#)
 - [Lambda discard parameters](#)
 - [Attributes on local functions](#)
- [Support for code generators](#)
 - [Module initializers](#)
 - [New features for partial methods](#)

C# 9.0 is supported on .NET 5. For more information, see [C# language versioning](#).

You can download the latest .NET SDK from the [.NET downloads page](#).

Record types

C# 9.0 introduces *record types*. You use the `record` keyword to define a reference type that provides built-in functionality for encapsulating data. You can create record types with immutable properties by using positional parameters or standard property syntax:

```
public record Person(string FirstName, string LastName);
```

```
public record Person
{
    public string FirstName { get; init; } = default!;
    public string LastName { get; init; } = default!;
};
```

You can also create record types with mutable properties and fields:

```
public record Person
{
    public string FirstName { get; set; } = default!;
    public string LastName { get; set; } = default!;
};
```

While records can be mutable, they are primarily intended for supporting immutable data models. The record type offers the following features:

- [Concise syntax for creating a reference type with immutable properties](#)
- Behavior useful for a data-centric reference type:
 - [Value equality](#)
 - [Concise syntax for nondestructive mutation](#)
 - [Built-in formatting for display](#)
- [Support for inheritance hierarchies](#)

You can use [structure types](#) to design data-centric types that provide value equality and little or no behavior. But for relatively large data models, structure types have some disadvantages:

- They don't support inheritance.
- They're less efficient at determining value equality. For value types, the [ValueType.Equals](#) method uses reflection to find all fields. For records, the compiler generates the `Equals` method. In practice, the implementation of value equality in records is measurably faster.
- They use more memory in some scenarios, since every instance has a complete copy of all of the data. Record types are [reference types](#), so a record instance contains only a reference to the data.

Positional syntax for property definition

You can use positional parameters to declare properties of a record and to initialize the property values when you create an instance:

```
public record Person(string FirstName, string LastName);

public static void Main()
{
    Person person = new("Nancy", "Davolio");
    Console.WriteLine(person);
    // output: Person { FirstName = Nancy, LastName = Davolio }
}
```

When you use the positional syntax for property definition, the compiler creates:

- A public init-only auto-implemented property for each positional parameter provided in the record declaration. An [init-only](#) property can only be set in the constructor or by using a property initializer.
- A primary constructor whose parameters match the positional parameters on the record declaration.
- A `Deconstruct` method with an `out` parameter for each positional parameter provided in the record declaration.

For more information, see [Positional syntax](#) in the C# language reference article about records.

Immutability

A record type is not necessarily immutable. You can declare properties with `set` accessors and fields that aren't `readonly`. But while records can be mutable, they make it easier to create immutable data models. Properties that you create by using positional syntax are immutable.

Immutability can be useful when you want a data-centric type to be thread-safe or a hash code to remain the

same in a hash table. It can prevent bugs that happen when you pass an argument by reference to a method, and the method unexpectedly changes the argument value.

The features unique to record types are implemented by compiler-synthesized methods, and none of these methods compromises immutability by modifying object state.

Value equality

Value equality means that two variables of a record type are equal if the types match and all property and field values match. For other reference types, equality means identity. That is, two variables of a reference type are equal if they refer to the same object.

The following example illustrates value equality of record types:

```
public record Person(string FirstName, string LastName, string[] PhoneNumbers);

public static void Main()
{
    var phoneNumbers = new string[2];
    Person person1 = new("Nancy", "Davolio", phoneNumbers);
    Person person2 = new("Nancy", "Davolio", phoneNumbers);
    Console.WriteLine(person1 == person2); // output: True

    person1.PhoneNumbers[0] = "555-1234";
    Console.WriteLine(person1 == person2); // output: True

    Console.WriteLine(ReferenceEquals(person1, person2)); // output: False
}
```

In `class` types, you could manually override equality methods and operators to achieve value equality, but developing and testing that code would be time-consuming and error-prone. Having this functionality built-in prevents bugs that would result from forgetting to update custom override code when properties or fields are added or changed.

For more information, see [Value equality](#) in the C# language reference article about records.

Nondestructive mutation

If you need to mutate immutable properties of a record instance, you can use a `with` expression to achieve *nondestructive mutation*. A `with` expression makes a new record instance that is a copy of an existing record instance, with specified properties and fields modified. You use [object initializer](#) syntax to specify the values to be changed, as shown in the following example:

```

public record Person(string FirstName, string LastName)
{
    public string[] PhoneNumbers { get; init; }
}

public static void Main()
{
    Person person1 = new("Nancy", "Davolio") { PhoneNumbers = new string[1] };
    Console.WriteLine(person1);
    // output: Person { FirstName = Nancy, LastName = Davolio, PhoneNumbers = System.String[] }

    Person person2 = person1 with { FirstName = "John" };
    Console.WriteLine(person2);
    // output: Person { FirstName = John, LastName = Davolio, PhoneNumbers = System.String[] }
    Console.WriteLine(person1 == person2); // output: False

    person2 = person1 with { PhoneNumbers = new string[1] };
    Console.WriteLine(person2);
    // output: Person { FirstName = Nancy, LastName = Davolio, PhoneNumbers = System.String[] }
    Console.WriteLine(person1 == person2); // output: False

    person2 = person1 with { };
    Console.WriteLine(person1 == person2); // output: True
}

```

For more information, see [Nondestructive mutation](#) in the C# language reference article about records.

Built-in formatting for display

Record types have a compiler-generated [ToString](#) method that displays the names and values of public properties and fields. The `ToString` method returns a string of the following format:

```
<record type name> { <property name> = <value>, <property name> = <value>, ...}
```

For reference types, the type name of the object that the property refers to is displayed instead of the property value. In the following example, the array is a reference type, so `System.String[]` is displayed instead of the actual array element values:

```
Person { FirstName = Nancy, LastName = Davolio, ChildNames = System.String[] }
```

For more information, see [Built-in formatting](#) in the C# language reference article about records.

Inheritance

A record can inherit from another record. However, a record can't inherit from a class, and a class can't inherit from a record.

The following example illustrates inheritance with positional property syntax:

```

public abstract record Person(string FirstName, string LastName);
public record Teacher(string FirstName, string LastName, int Grade)
    : Person(FirstName, LastName);
public static void Main()
{
    Person teacher = new Teacher("Nancy", "Davolio", 3);
    Console.WriteLine(teacher);
    // output: Teacher { FirstName = Nancy, LastName = Davolio, Grade = 3 }
}

```

For two record variables to be equal, the run-time type must be equal. The types of the containing variables might be different. This is illustrated in the following code example:

```

public abstract record Person(string FirstName, string LastName);
public record Teacher(string FirstName, string LastName, int Grade)
    : Person(FirstName, LastName);
public record Student(string FirstName, string LastName, int Grade)
    : Person(FirstName, LastName);
public static void Main()
{
    Person teacher = new Teacher("Nancy", "Davolio", 3);
    Person student = new Student("Nancy", "Davolio", 3);
    Console.WriteLine(teacher == student); // output: False

    Student student2 = new Student("Nancy", "Davolio", 3);
    Console.WriteLine(student2 == student); // output: True
}

```

In the example, all instances have the same properties and the same property values. But `student == teacher` returns `False` although both are `Person`-type variables. And `student == student2` returns `True` although one is a `Person` variable and one is a `Student` variable.

All public properties and fields of both derived and base types are included in the `ToString` output, as shown in the following example:

```

public abstract record Person(string FirstName, string LastName);
public record Teacher(string FirstName, string LastName, int Grade)
    : Person(FirstName, LastName);
public record Student(string FirstName, string LastName, int Grade)
    : Person(FirstName, LastName);

public static void Main()
{
    Person teacher = new Teacher("Nancy", "Davolio", 3);
    Console.WriteLine(teacher);
    // output: Teacher { FirstName = Nancy, LastName = Davolio, Grade = 3 }
}

```

For more information, see [Inheritance](#) in the C# language reference article about records.

Init only setters

Init only setters provide consistent syntax to initialize members of an object. Property initializers make it clear which value is setting which property. The downside is that those properties must be settable. Starting with C# 9.0, you can create `init` accessors instead of `set` accessors for properties and indexers. Callers can use property initializer syntax to set these values in creation expressions, but those properties are readonly once construction has completed. Init only setters provide a window to change state. That window closes when the construction phase ends. The construction phase effectively ends after all initialization, including property initializers and with-expressions have completed.

You can declare `init` only setters in any type you write. For example, the following struct defines a weather observation structure:


```
public struct WeatherObservation
{
    public DateTime RecordedAt { get; init; }
    public decimal TemperatureInCelsius { get; init; }
    public decimal PressureInMillibars { get; init; }

    public override string ToString() =>
        $"At {RecordedAt:h:mm tt} on {RecordedAt:M/d/yyyy}: " +
        $"Temp = {TemperatureInCelsius}, with {PressureInMillibars} pressure";
}
```

Callers can use property initializer syntax to set the values, while still preserving the immutability:

```
var now = new WeatherObservation
{
    RecordedAt = DateTime.Now,
    TemperatureInCelsius = 20,
    PressureInMillibars = 998.0m
};
```

An attempt to change an observation after initialization results in a compiler error:

```
// Error! CS8852.
now.TemperatureInCelsius = 18;
```

Init only setters can be useful to set base class properties from derived classes. They can also set derived properties through helpers in a base class. Positional records declare properties using init only setters. Those setters are used in with-expressions. You can declare init only setters for any `class`, `struct`, or `record` you define.

For more information, see [init \(C# Reference\)](#).

Top-level statements

Top-level statements remove unnecessary ceremony from many applications. Consider the canonical "Hello World!" program:

```
using System;

namespace HelloWorld
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Hello World!");
        }
    }
}
```

There's only one line of code that does anything. With top-level statements, you can replace all that boilerplate with the `using` directive and the single line that does the work:

```
using System;

Console.WriteLine("Hello World!");
```

If you wanted a one-line program, you could remove the `using` directive and use the fully qualified type name:

```
System.Console.WriteLine("Hello World!");
```

Only one file in your application may use top-level statements. If the compiler finds top-level statements in multiple source files, it's an error. It's also an error if you combine top-level statements with a declared program entry point method, typically a `Main` method. In a sense, you can think that one file contains the statements that would normally be in the `Main` method of a `Program` class.

One of the most common uses for this feature is creating teaching materials. Beginner C# developers can write the canonical "Hello World!" in one or two lines of code. None of the extra ceremony is needed. However, seasoned developers will find many uses for this feature as well. Top-level statements enable a script-like experience for experimentation similar to what Jupyter notebooks provide. Top-level statements are great for small console programs and utilities. [Azure Functions](#) is an ideal use case for top-level statements.

Most importantly, top-level statements don't limit your application's scope or complexity. Those statements can access or use any .NET class. They also don't limit your use of command-line arguments or return values. Top-level statements can access an array of strings named `args`. If the top-level statements return an integer value, that value becomes the integer return code from a synthesized `Main` method. The top-level statements may contain async expressions. In that case, the synthesized entry point returns a `Task`, or `Task<int>`.

For more information, see [Top-level statements](#) in the C# Programming Guide.

Pattern matching enhancements

C# 9 includes new pattern matching improvements:

- **Type patterns** match a variable is a type
- **Parenthesized patterns** enforce or emphasize the precedence of pattern combinations
- **Conjunctive `and` patterns** require both patterns to match
- **Disjunctive `or` patterns** require either pattern to match
- **Negated `not` patterns** require that a pattern doesn't match
- **Relational patterns** require the input be less than, greater than, less than or equal, or greater than or equal to a given constant.

These patterns enrich the syntax for patterns. Consider these examples:

```
public static bool IsLetter(this char c) =>
    c is >= 'a' and <= 'z' or >= 'A' and <= 'Z';
```

With optional parentheses to make it clear that `and` has higher precedence than `or`:

```
public static bool IsLetterOrSeparator(this char c) =>
    c is (>= 'a' and <= 'z') or (>= 'A' and <= 'Z') or '.' or ',';
```

One of the most common uses is a new syntax for a null check:

```
if (e is not null)
{
    // ...
}
```

Any of these patterns can be used in any context where patterns are allowed: `is` pattern expressions, `switch`

expressions, nested patterns, and the pattern of a `switch` statement's `case` label.

For more information, see [Patterns \(C# reference\)](#).

For more information, see the [Relational patterns](#) and [Logical patterns](#) sections of the [Patterns](#) article.

Performance and interop

Three new features improve support for native interop and low-level libraries that require high performance: native sized integers, function pointers, and omitting the `localsinit` flag.

Native sized integers, `nint` and `nuint`, are integer types. They're expressed by the underlying types [System.IntPtr](#) and [System.UIntPtr](#). The compiler surfaces additional conversions and operations for these types as native ints. Native sized integers define properties for `MaxValue` or `MinValue`. These values can't be expressed as compile-time constants because they depend on the native size of an integer on the target machine. Those values are readonly at run time. You can use constant values for `nint` in the range [`int.MinValue` .. `int.MaxValue`]. You can use constant values for `nuint` in the range [`uint.MinValue` .. `uint.MaxValue`]. The compiler performs constant folding for all unary and binary operators using the [System.Int32](#) and [System.UInt32](#) types. If the result doesn't fit in 32 bits, the operation is executed at run time and isn't considered a constant. Native sized integers can increase performance in scenarios where integer math is used extensively and needs to have the fastest performance possible. For more information, see [nint and nuint types](#)

Function pointers provide an easy syntax to access the IL opcodes `ldftn` and `calli`. You can declare function pointers using new `delegate*` syntax. A `delegate*` type is a pointer type. Invoking the `delegate*` type uses `calli`, in contrast to a delegate that uses `callvirt` on the `Invoke()` method. Syntactically, the invocations are identical. Function pointer invocation uses the `managed` calling convention. You add the `unmanaged` keyword after the `delegate*` syntax to declare that you want the `unmanaged` calling convention. Other calling conventions can be specified using attributes on the `delegate*` declaration. For more information, see [Unsafe code and pointer types](#).

Finally, you can add the [System.Runtime.CompilerServices.SkipLocalsInitAttribute](#) to instruct the compiler not to emit the `localsinit` flag. This flag instructs the CLR to zero-initialize all local variables. The `localsinit` flag has been the default behavior for C# since 1.0. However, the extra zero-initialization may have measurable performance impact in some scenarios. In particular, when you use `stackalloc`. In those cases, you can add the [SkipLocalsInitAttribute](#). You may add it to a single method or property, or to a `class`, `struct`, `interface`, or even a module. This attribute doesn't affect `abstract` methods; it affects the code generated for the implementation. For more information, see [SkipLocalsInit attribute](#).

These features can improve performance in some scenarios. They should be used only after careful benchmarking both before and after adoption. Code involving native sized integers must be tested on multiple target platforms with different integer sizes. The other features require unsafe code.

Fit and finish features

Many of the other features help you write code more efficiently. In C# 9.0, you can omit the type in a `new` expression when the created object's type is already known. The most common use is in field declarations:

```
private List<WeatherObservation> _observations = new();
```

Target-typed `new` can also be used when you need to create a new object to pass as an argument to a method. Consider a `ForecastFor()` method with the following signature:

```
public WeatherForecast ForecastFor(DateTime forecastDate, WeatherForecastOptions options)
```

You could call it as follows:

```
var forecast = station.ForecastFor(DateTime.Now.AddDays(2), new());
```

Another nice use for this feature is to combine it with init only properties to initialize a new object:

```
WeatherStation station = new() { Location = "Seattle, WA" };
```

You can return an instance created by the default constructor using a `return new();` statement.

A similar feature improves the target type resolution of [conditional expressions](#). With this change, the two expressions need not have an implicit conversion from one to the other, but may both have implicit conversions to a target type. You likely won't notice this change. What you will notice is that some conditional expressions that previously required casts or wouldn't compile now just work.

Starting in C# 9.0, you can add the `static` modifier to [lambda expressions](#) or [anonymous methods](#). Static lambda expressions are analogous to the `static` local functions: a static lambda or anonymous method can't capture local variables or instance state. The `static` modifier prevents accidentally capturing other variables.

Covariant return types provide flexibility for the return types of [override](#) methods. An override method can return a type derived from the return type of the overridden base method. This can be useful for records and for other types that support virtual clone or factory methods.

In addition, the `foreach` loop will recognize and use an extension method `GetEnumerator` that otherwise satisfies the `foreach` pattern. This change means `foreach` is consistent with other pattern-based constructions such as the `async` pattern, and pattern-based deconstruction. In practice, this change means you can add `foreach` support to any type. You should limit its use to when enumerating an object makes sense in your design.

Next, you can use discards as parameters to lambda expressions. This convenience enables you to avoid naming the argument, and the compiler may avoid using it. You use the `_` for any argument. For more information, see the [Input parameters of a lambda expression](#) section of the [Lambda expressions](#) article.

Finally, you can now apply attributes to [local functions](#). For example, you can apply [nullable attribute annotations](#) to local functions.

Support for code generators

Two final features support C# code generators. C# code generators are a component you can write that is similar to a Roslyn analyzer or code fix. The difference is that code generators analyze code and write new source code files as part of the compilation process. A typical code generator searches code for attributes or other conventions.

A code generator reads attributes or other code elements using the Roslyn analysis APIs. From that information, it adds new code to the compilation. Source generators can only add code; they aren't allowed to modify any existing code in the compilation.

The two features added for code generators are extensions to *partial method syntax*, and *module initializers*. First, the changes to partial methods. Before C# 9.0, partial methods are `private` but can't specify an access modifier, have a `void` return, and can't have `out` parameters. These restrictions meant that if no method implementation is provided, the compiler removes all calls to the partial method. C# 9.0 removes these restrictions, but requires that partial method declarations have an implementation. Code generators can provide

that implementation. To avoid introducing a breaking change, the compiler considers any partial method without an access modifier to follow the old rules. If the partial method includes the `private` access modifier, the new rules govern that partial method. For more information, see [partial method \(C# Reference\)](#).

The second new feature for code generators is *module initializers*. Module initializers are methods that have the [ModuleInitializerAttribute](#) attribute attached to them. These methods will be called by the runtime before any other field access or method invocation within the entire module. A module initializer method:

- Must be static
- Must be parameterless
- Must return void
- Must not be a generic method
- Must not be contained in a generic class
- Must be accessible from the containing module

That last bullet point effectively means the method and its containing class must be internal or public. The method can't be a local function. For more information, see [ModuleInitializer attribute](#).

What's new in C# 8.0

12/28/2021 • 16 minutes to read • [Edit Online](#)

C# 8.0 adds the following features and enhancements to the C# language:

- [Readonly members](#)
- [Default interface methods](#)
- [Pattern matching enhancements](#):
 - [Switch expressions](#)
 - [Property patterns](#)
 - [Tuple patterns](#)
 - [Positional patterns](#)
- [Using declarations](#)
- [Static local functions](#)
- [Disposable ref structs](#)
- [Nullable reference types](#)
- [Asynchronous streams](#)
- [Asynchronous disposable](#)
- [Indices and ranges](#)
- [Null-coalescing assignment](#)
- [Unmanaged constructed types](#)
- [Stackalloc in nested expressions](#)
- [Enhancement of interpolated verbatim strings](#)

C# 8.0 is supported on **.NET Core 3.x** and **.NET Standard 2.1**. For more information, see [C# language versioning](#).

The remainder of this article briefly describes these features. Where in-depth articles are available, links to those tutorials and overviews are provided. You can explore these features in your environment using the `dotnet try` global tool:

1. Install the [dotnet-try](#) global tool.
2. Clone the [dotnet/try-samples](#) repository.
3. Set the current directory to the *csharp8* subdirectory for the *try-samples* repository.
4. Run `dotnet try`.

Readonly members

You can apply the `readonly` modifier to members of a struct. It indicates that the member doesn't modify state. It's more granular than applying the `readonly` modifier to a `struct` declaration. Consider the following mutable struct:

```
public struct Point
{
    public double X { get; set; }
    public double Y { get; set; }
    public double Distance => Math.Sqrt(X * X + Y * Y);

    public override string ToString() =>
        $"({X}, {Y}) is {Distance} from the origin";
}
```

Like most structs, the `ToString()` method doesn't modify state. You could indicate that by adding the `readonly` modifier to the declaration of `ToString()`:

```
public readonly override string ToString() =>
    $"({X}, {Y}) is {Distance} from the origin";
```

The preceding change generates a compiler warning, because `ToString` accesses the `Distance` property, which isn't marked `readonly`:

```
warning CS8656: Call to non-readonly member 'Point.Distance.get' from a 'readonly' member results in an
implicit copy of 'this'
```

The compiler warns you when it needs to create a defensive copy. The `Distance` property doesn't change state, so you can fix this warning by adding the `readonly` modifier to the declaration:

```
public readonly double Distance => Math.Sqrt(X * X + Y * Y);
```

Notice that the `readonly` modifier is necessary on a read-only property. The compiler doesn't assume `get` accessors don't modify state; you must declare `readonly` explicitly. Auto-implemented properties are an exception; the compiler will treat all auto-implemented getters as `readonly`, so here there's no need to add the `readonly` modifier to the `x` and `y` properties.

The compiler does enforce the rule that `readonly` members don't modify state. The following method won't compile unless you remove the `readonly` modifier:

```
public readonly void Translate(int xOffset, int yOffset)
{
    X += xOffset;
    Y += yOffset;
}
```

This feature lets you specify your design intent so the compiler can enforce it, and make optimizations based on that intent.

For more information, see the [readonly instance members](#) section of the [Structure types](#) article.

Default interface methods

You can now add members to interfaces and provide an implementation for those members. This language feature enables API authors to add methods to an interface in later versions without breaking source or binary compatibility with existing implementations of that interface. Existing implementations *inherit* the default implementation. This feature also enables C# to interoperate with APIs that target Android or Swift, which support similar features. Default interface methods also enable scenarios similar to a "traits" language feature.

Default interface methods affect many scenarios and language elements. Our first tutorial covers [updating an interface with default implementations](#).

More patterns in more places

Pattern matching gives tools to provide shape-dependent functionality across related but different kinds of data. C# 7.0 introduced syntax for type patterns and constant patterns by using the `is` expression and the `switch` statement. These features represented the first tentative steps toward supporting programming paradigms where data and functionality live apart. As the industry moves toward more microservices and other cloud-based architectures, other language tools are needed.

C# 8.0 expands this vocabulary so you can use more pattern expressions in more places in your code. Consider these features when your data and functionality are separate. Consider pattern matching when your algorithms depend on a fact other than the runtime type of an object. These techniques provide another way to express designs.

In addition to new patterns in new places, C# 8.0 adds **recursive patterns**. Recursive patterns are patterns that can contain other patterns.

Switch expressions

Often, a `switch` statement produces a value in each of its `case` blocks. **Switch expressions** enable you to use more concise expression syntax. There are fewer repetitive `case` and `break` keywords, and fewer curly braces. As an example, consider the following enum that lists the colors of the rainbow:

```
public enum Rainbow
{
    Red,
    Orange,
    Yellow,
    Green,
    Blue,
    Indigo,
    Violet
}
```

If your application defined an `RGBColor` type that is constructed from the `R`, `G` and `B` components, you could convert a `Rainbow` value to its RGB values using the following method containing a switch expression:

```
public static RGBColor FromRainbow(Rainbow colorBand) =>
    colorBand switch
    {
        Rainbow.Red    => new RGBColor(0xFF, 0x00, 0x00),
        Rainbow.Orange => new RGBColor(0xFF, 0x7F, 0x00),
        Rainbow.Yellow => new RGBColor(0xFF, 0xFF, 0x00),
        Rainbow.Green  => new RGBColor(0x00, 0xFF, 0x00),
        Rainbow.Blue   => new RGBColor(0x00, 0x00, 0xFF),
        Rainbow.Indigo => new RGBColor(0x4B, 0x00, 0x82),
        Rainbow.Violet => new RGBColor(0x94, 0x00, 0xD3),
        _               => throw new ArgumentException(message: "invalid enum value", paramName:
        nameof(colorBand)),
    };
```

There are several syntax improvements here:

- The variable comes before the `switch` keyword. The different order makes it visually easy to distinguish the switch expression from the switch statement.
- The `case` and `:` elements are replaced with `=>`. It's more concise and intuitive.

- The `default` case is replaced with a `_` discard.
- The bodies are expressions, not statements.

Contrast that with the equivalent code using the classic `switch` statement:

```
public static RGBColor FromRainbowClassic(Rainbow colorBand)
{
    switch (colorBand)
    {
        case Rainbow.Red:
            return new RGBColor(0xFF, 0x00, 0x00);
        case Rainbow.Orange:
            return new RGBColor(0xFF, 0x7F, 0x00);
        case Rainbow.Yellow:
            return new RGBColor(0xFF, 0xFF, 0x00);
        case Rainbow.Green:
            return new RGBColor(0x00, 0xFF, 0x00);
        case Rainbow.Blue:
            return new RGBColor(0x00, 0x00, 0xFF);
        case Rainbow.Indigo:
            return new RGBColor(0x4B, 0x00, 0x82);
        case Rainbow.Violet:
            return new RGBColor(0x94, 0x00, 0xD3);
        default:
            throw new ArgumentException(message: "invalid enum value", paramName: nameof(colorBand));
    };
}
```

For more information, see [switch expression](#).

Property patterns

The **property pattern** enables you to match on properties of the object examined. Consider an eCommerce site that must compute sales tax based on the buyer's address. That computation isn't a core responsibility of an `Address` class. It will change over time, likely more often than address format changes. The amount of sales tax depends on the `State` property of the address. The following method uses the property pattern to compute the sales tax from the address and the price:

```
public static decimal ComputeSalesTax(Address location, decimal salePrice) =>
    location switch
    {
        { State: "WA" } => salePrice * 0.06M,
        { State: "MN" } => salePrice * 0.075M,
        { State: "MI" } => salePrice * 0.05M,
        // other cases removed for brevity...
        _ => 0M
    };

```

Pattern matching creates a concise syntax for expressing this algorithm.

For more information, see the [Property pattern](#) section of the [Patterns](#) article.

Tuple patterns

Some algorithms depend on multiple inputs. **Tuple patterns** allow you to switch based on multiple values expressed as a [tuple](#). The following code shows a switch expression for the game *rock, paper, scissors*.

```

public static string RockPaperScissors(string first, string second)
    => (first, second) switch
    {
        ("rock", "paper") => "rock is covered by paper. Paper wins.",
        ("rock", "scissors") => "rock breaks scissors. Rock wins.",
        ("paper", "rock") => "paper covers rock. Paper wins.",
        ("paper", "scissors") => "paper is cut by scissors. Scissors wins.",
        ("scissors", "rock") => "scissors is broken by rock. Rock wins.",
        ("scissors", "paper") => "scissors cuts paper. Scissors wins.",
        (_, _) => "tie"
    };

```

The messages indicate the winner. The discard case represents the three combinations for ties, or other text inputs.

Positional patterns

Some types include a `Deconstruct` method that deconstructs its properties into discrete variables. When a `Deconstruct` method is accessible, you can use **positional patterns** to inspect properties of the object and use those properties for a pattern. Consider the following `Point` class that includes a `Deconstruct` method to create discrete variables for `x` and `y`:

```

public class Point
{
    public int X { get; }
    public int Y { get; }

    public Point(int x, int y) => (X, Y) = (x, y);

    public void Deconstruct(out int x, out int y) =>
        (x, y) = (X, Y);
}

```

Additionally, consider the following enum that represents various positions of a quadrant:

```

public enum Quadrant
{
    Unknown,
    Origin,
    One,
    Two,
    Three,
    Four,
    OnBorder
}

```

The following method uses the **positional pattern** to extract the values of `x` and `y`. Then, it uses a `when` clause to determine the `Quadrant` of the point:

```

static Quadrant GetQuadrant(Point point) => point switch
{
    (0, 0) => Quadrant.Origin,
    var (x, y) when x > 0 && y > 0 => Quadrant.One,
    var (x, y) when x < 0 && y > 0 => Quadrant.Two,
    var (x, y) when x < 0 && y < 0 => Quadrant.Three,
    var (x, y) when x > 0 && y < 0 => Quadrant.Four,
    var (_, _) => Quadrant.OnBorder,
    _ => Quadrant.Unknown
};

```

The discard pattern in the preceding switch matches when either `x` or `y` is 0, but not both. A switch expression must either produce a value or throw an exception. If none of the cases match, the switch expression throws an exception. The compiler generates a warning for you if you don't cover all possible cases in your switch expression.

You can explore pattern matching techniques in this [advanced tutorial on pattern matching](#). For more information about a positional pattern, see the [Positional pattern](#) section of the [Patterns](#) article.

Using declarations

A **using declaration** is a variable declaration preceded by the `using` keyword. It tells the compiler that the variable being declared should be disposed at the end of the enclosing scope. For example, consider the following code that writes a text file:

```
static int WriteLinesToFile(IEnumerable<string> lines)
{
    using var file = new System.IO.StreamWriter("WriteLines2.txt");
    int skippedLines = 0;
    foreach (string line in lines)
    {
        if (!line.Contains("Second"))
        {
            file.WriteLine(line);
        }
        else
        {
            skippedLines++;
        }
    }
    // Notice how skippedLines is in scope here.
    return skippedLines;
    // file is disposed here
}
```

In the preceding example, the file is disposed when the closing brace for the method is reached. That's the end of the scope in which `file` is declared. The preceding code is equivalent to the following code that uses the classic [using statement](#):

```
static int WriteLinesToFile(IEnumerable<string> lines)
{
    using (var file = new System.IO.StreamWriter("WriteLines2.txt"))
    {
        int skippedLines = 0;
        foreach (string line in lines)
        {
            if (!line.Contains("Second"))
            {
                file.WriteLine(line);
            }
            else
            {
                skippedLines++;
            }
        }
        return skippedLines;
    } // file is disposed here
}
```

In the preceding example, the file is disposed when the closing brace associated with the `using` statement is reached.

In both cases, the compiler generates the call to `Dispose()`. The compiler generates an error if the expression in the `using` statement isn't disposable.

Static local functions

You can now add the `static` modifier to [local functions](#) to ensure that local function doesn't capture (reference) any variables from the enclosing scope. Doing so generates `CS8421`, "A static local function can't contain a reference to <variable>."

Consider the following code. The local function `LocalFunction` accesses the variable `y`, declared in the enclosing scope (the method `M`). Therefore, `LocalFunction` can't be declared with the `static` modifier:

```
int M()
{
    int y;
    LocalFunction();
    return y;

    void LocalFunction() => y = 0;
}
```

The following code contains a static local function. It can be static because it doesn't access any variables in the enclosing scope:

```
int M()
{
    int y = 5;
    int x = 7;
    return Add(x, y);

    static int Add(int left, int right) => left + right;
}
```

Disposable ref structs

A `struct` declared with the `ref` modifier may not implement any interfaces and so can't implement [IDisposable](#). Therefore, to enable a `ref struct` to be disposed, it must have an accessible `void Dispose()` method. This feature also applies to `readonly ref struct` declarations.

Nullable reference types

Inside a nullable annotation context, any variable of a reference type is considered to be a **nonnullable reference type**. If you want to indicate that a variable may be null, you must append the type name with the `?` to declare the variable as a **nullable reference type**.

For nonnullable reference types, the compiler uses flow analysis to ensure that local variables are initialized to a non-null value when declared. Fields must be initialized during construction. The compiler generates a warning if the variable isn't set by a call to any of the available constructors or by an initializer. Furthermore, nonnullable reference types can't be assigned a value that could be null.

Nullable reference types aren't checked to ensure they aren't assigned or initialized to null. However, the compiler uses flow analysis to ensure that any variable of a nullable reference type is checked against null before it's accessed or assigned to a nonnullable reference type.

You can learn more about the feature in the overview of [nullable reference types](#). Try it yourself in a new application in this [nullable reference types tutorial](#). Learn about the steps to migrate an existing codebase to

make use of nullable reference types in the article on [upgrading to nullable reference types](#).

Asynchronous streams

Starting with C# 8.0, you can create and consume streams asynchronously. A method that returns an asynchronous stream has three properties:

1. It's declared with the `async` modifier.
2. It returns an `IAsyncEnumerable<T>`.
3. The method contains `yield return` statements to return successive elements in the asynchronous stream.

Consuming an asynchronous stream requires you to add the `await` keyword before the `foreach` keyword when you enumerate the elements of the stream. Adding the `await` keyword requires the method that enumerates the asynchronous stream to be declared with the `async` modifier and to return a type allowed for an `async` method. Typically that means returning a `Task` or `Task<TResult>`. It can also be a `ValueTask` or `ValueTask<TResult>`. A method can both consume and produce an asynchronous stream, which means it would return an `IAsyncEnumerable<T>`. The following code generates a sequence from 0 to 19, waiting 100 ms between generating each number:

```
public static async System.Collections.Generic.IAsyncEnumerable<int> GenerateSequence()
{
    for (int i = 0; i < 20; i++)
    {
        await Task.Delay(100);
        yield return i;
    }
}
```

You would enumerate the sequence using the `await foreach` statement:

```
await foreach (var number in GenerateSequence())
{
    Console.WriteLine(number);
}
```

You can try asynchronous streams yourself in our tutorial on [creating and consuming async streams](#). By default, stream elements are processed in the captured context. If you want to disable capturing of the context, use the `TaskAsyncEnumerableExtensions.ConfigureAwait` extension method. For more information about synchronization contexts and capturing the current context, see the article on [consuming the Task-based asynchronous pattern](#).

Asynchronous disposable

Starting with C# 8.0, the language supports asynchronous disposable types that implement the `System.IAsyncDisposable` interface. You use the `await using` statement to work with an asynchronously disposable object. For more information, see the [Implement a DisposeAsync method](#) article.

Indices and ranges

Indices and ranges provide a succinct syntax for accessing single elements or ranges in a sequence.

This language support relies on two new types, and two new operators:

- `System.Index` represents an index into a sequence.
- The index from end operator `^`, which specifies that an index is relative to the end of the sequence.

- `System.Range` represents a sub range of a sequence.
- The range operator `..`, which specifies the start and end of a range as its operands.

Let's start with the rules for indexes. Consider an array `sequence`. The `0` index is the same as `sequence[0]`. The `^0` index is the same as `sequence[sequence.Length]`. Note that `sequence[^0]` does throw an exception, just as `sequence[sequence.Length]` does. For any number `n`, the index `^n` is the same as `sequence.Length - n`.

A range specifies the *start* and *end* of a range. The start of the range is inclusive, but the end of the range is exclusive, meaning the *start* is included in the range but the *end* isn't included in the range. The range `[0..^0]` represents the entire range, just as `[0..sequence.Length]` represents the entire range.

Let's look at a few examples. Consider the following array, annotated with its index from the start and from the end:

```
var words = new string[]
{
    "The",           // index from start  index from end
    "quick",         // 0                ^9
    "brown",         // 1                ^8
    "fox",           // 2                ^7
    "jumped",        // 3                ^6
    "over",          // 4                ^5
    "the",           // 5                ^4
    "lazy",          // 6                ^3
    "dog",           // 7                ^2
    "dog"            // 8                ^1
};                  // 9 (or words.Length) ^0
```

You can retrieve the last word with the `^1` index:

```
Console.WriteLine($"The last word is {words[^1]}");
// writes "dog"
```

The following code creates a subrange with the words "quick", "brown", and "fox". It includes `words[1]` through `words[3]`. The element `words[4]` isn't in the range.

```
var quickBrownFox = words[1..4];
```

The following code creates a subrange with "lazy" and "dog". It includes `words[^2]` and `words[^1]`. The end index `words[^0]` isn't included:

```
var lazyDog = words[^2..^0];
```

The following examples create ranges that are open ended for the start, end, or both:

```
var allWords = words[..]; // contains "The" through "dog".
var firstPhrase = words[..4]; // contains "The" through "fox"
var lastPhrase = words[6..]; // contains "the", "lazy" and "dog"
```

You can also declare ranges as variables:

```
Range phrase = 1..4;
```

The range can then be used inside the `[` and `]` characters:

```
var text = words[phrase];
```

Not only arrays support indices and ranges. You can also use indices and ranges with [string](#), [Span<T>](#), or [ReadOnlySpan<T>](#). For more information, see [Type support for indices and ranges](#).

You can explore more about indices and ranges in the tutorial on [indices and ranges](#).

Null-coalescing assignment

C# 8.0 introduces the null-coalescing assignment operator `??=`. You can use the `??=` operator to assign the value of its right-hand operand to its left-hand operand only if the left-hand operand evaluates to `null`.

```
List<int> numbers = null;
int? i = null;

numbers ??= new List<int>();
numbers.Add(i ??= 17);
numbers.Add(i ??= 20);

Console.WriteLine(string.Join(" ", numbers)); // output: 17 17
Console.WriteLine(i); // output: 17
```

For more information, see the [?? and ??= operators](#) article.

Unmanaged constructed types

In C# 7.3 and earlier, a constructed type (a type that includes at least one type argument) can't be an [unmanaged type](#). Starting with C# 8.0, a constructed value type is unmanaged if it contains fields of unmanaged types only.

For example, given the following definition of the generic `Coords<T>` type:

```
public struct Coords<T>
{
    public T X;
    public T Y;
}
```

the `Coords<int>` type is an unmanaged type in C# 8.0 and later. Like for any unmanaged type, you can create a pointer to a variable of this type or [allocate a block of memory on the stack](#) for instances of this type:

```
Span<Coords<int>> coordinates = stackalloc[]
{
    new Coords<int> { X = 0, Y = 0 },
    new Coords<int> { X = 0, Y = 3 },
    new Coords<int> { X = 4, Y = 0 }
};
```

For more information, see [Unmanaged types](#).

Stackalloc in nested expressions

Starting with C# 8.0, if the result of a `stackalloc` expression is of the [System.Span<T>](#) or [System.ReadOnlySpan<T>](#) type, you can use the `stackalloc` expression in other expressions:

```
Span<int> numbers = stackalloc[] { 1, 2, 3, 4, 5, 6 };  
var ind = numbers.IndexOfAny(stackalloc[] { 2, 4, 6, 8 });  
Console.WriteLine(ind); // output: 1
```

Enhancement of interpolated verbatim strings

Order of the `$` and `@` tokens in [interpolated](#) verbatim strings can be any: both `$$"..."` and `@$"..."` are valid interpolated verbatim strings. In earlier C# versions, the `$` token must appear before the `@` token.

Learn about any breaking changes in the C# compiler

12/28/2021 • 2 minutes to read • [Edit Online](#)

The [Roslyn](#) team maintains a list of breaking changes in the C# and Visual Basic compilers. You can find information on those changes at these links on their GitHub repository:

- [Breaking changes in Roslyn in after .NET 6 all the way to .NET 7](#)
- [Breaking changes in Roslyn in C# 10.0/.NET 6](#)
- [Breaking changes in Roslyn after .NET 5](#)
- [Breaking changes in VS2019 version 16.8 introduced with .NET 5 and C# 9.0](#)
- [Breaking changes in VS2019 Update 1 and beyond compared to VS2019](#)
- [Breaking changes since VS2017 \(C# 7\)](#)
- [Breaking changes in Roslyn 3.0 \(VS2019\) from Roslyn 2.* \(VS2017\)](#)
- [Breaking changes in Roslyn 2.0 \(VS2017\) from Roslyn 1.* \(VS2015\) and native C# compiler \(VS2013 and previous\).](#)
- [Breaking changes in Roslyn 1.0 \(VS2015\) from the native C# compiler \(VS2013 and previous\).](#)
- [Unicode version change in C# 6](#)

The history of C#

12/28/2021 • 13 minutes to read • [Edit Online](#)

This article provides a history of each major release of the C# language. The C# team is continuing to innovate and add new features. Detailed language feature status, including features considered for upcoming releases can be found [on the dotnet/roslyn repository](#) on GitHub.

IMPORTANT

The C# language relies on types and methods in what the C# specification defines as a *standard library* for some of the features. The .NET platform delivers those types and methods in a number of packages. One example is exception processing. Every `throw` statement or expression is checked to ensure the object being thrown is derived from [Exception](#). Similarly, every `catch` is checked to ensure that the type being caught is derived from [Exception](#). Each version may add new requirements. To use the latest language features in older environments, you may need to install specific libraries. These dependencies are documented in the page for each specific version. You can learn more about the [relationships between language and library](#) for background on this dependency.

C# version 1.0

When you go back and look, C# version 1.0, released with Visual Studio .NET 2002, looked a lot like Java. As [part of its stated design goals for ECMA](#), it sought to be a "simple, modern, general-purpose object-oriented language." At the time, looking like Java meant it achieved those early design goals.

But if you look back on C# 1.0 now, you'd find yourself a little dizzy. It lacked the built-in async capabilities and some of the slick functionality around generics you take for granted. As a matter of fact, it lacked generics altogether. And [LINQ](#)? Not available yet. Those additions would take some years to come out.

C# version 1.0 looked stripped of features, compared to today. You'd find yourself writing some verbose code. But yet, you have to start somewhere. C# version 1.0 was a viable alternative to Java on the Windows platform.

The major features of C# 1.0 included:

- [Classes](#)
- [Structs](#)
- [Interfaces](#)
- [Events](#)
- [Properties](#)
- [Delegates](#)
- [Operators and expressions](#)
- [Statements](#)
- [Attributes](#)

C# version 1.2

C# version 1.2 shipped with Visual Studio .NET 2003. It contained a few small enhancements to the language. Most notable is that starting with this version, the code generated in a `foreach` loop called [Dispose](#) on an [IEnumerator](#) when that [IEnumerator](#) implemented [IDisposable](#).

C# version 2.0

Now things start to get interesting. Let's take a look at some major features of C# 2.0, released in 2005, along with Visual Studio 2005:

- [Generics](#)
- [Partial types](#)
- [Anonymous methods](#)
- [Nullable value types](#)
- [Iterators](#)
- [Covariance and contravariance](#)

Other C# 2.0 features added capabilities to existing features:

- Getter/setter separate accessibility
- Method group conversions (delegates)
- Static classes
- Delegate inference

While C# may have started as a generic Object-Oriented (OO) language, C# version 2.0 changed that in a hurry. Once they had their feet under them, they went after some serious developer pain points. And they went after them in a significant way.

With generics, types and methods can operate on an arbitrary type while still retaining type safety. For instance, having a `List<T>` lets you have `List<string>` or `List<int>` and perform type-safe operations on those strings or integers while you iterate through them. Using generics is better than creating a `ListInt` type that derives from `ArrayList` or casting from `Object` for every operation.

C# version 2.0 brought iterators. To put it succinctly, iterators let you examine all the items in a `List` (or other Enumerable types) with a `foreach` loop. Having iterators as a first-class part of the language dramatically enhanced readability of the language and people's ability to reason about the code.

And yet, C# continued to play a bit of catch-up with Java. Java had already released versions that included generics and iterators. But that would soon change as the languages continued to evolve apart.

C# version 3.0

C# version 3.0 came in late 2007, along with Visual Studio 2008, though the full boat of language features would actually come with .NET Framework version 3.5. This version marked a major change in the growth of C#. It established C# as a truly formidable programming language. Let's take a look at some major features in this version:

- [Auto-implemented properties](#)
- [Anonymous types](#)
- [Query expressions](#)
- [Lambda expressions](#)
- [Expression trees](#)
- [Extension methods](#)
- [Implicitly typed local variables](#)
- [Partial methods](#)
- [Object and collection initializers](#)

In retrospect, many of these features seem both inevitable and inseparable. They all fit together strategically. It's thought that C# version's killer feature was the query expression, also known as Language-Integrated Query (LINQ).

A more nuanced view examines expression trees, lambda expressions, and anonymous types as the foundation upon which LINQ is constructed. But, in either case, C# 3.0 presented a revolutionary concept. C# 3.0 had begun to lay the groundwork for turning C# into a hybrid Object-Oriented / Functional language.

Specifically, you could now write SQL-style, declarative queries to perform operations on collections, among other things. Instead of writing a `for` loop to compute the average of a list of integers, you could now do that as simply as `list.Average()`. The combination of query expressions and extension methods made it look as though that list of integers had gotten a whole lot smarter.

It took time for people to really grasp and integrate the concept, but they gradually did. And now, years later, code is much more concise, simple, and functional.

C# version 4.0

C# version 4.0, released with Visual Studio 2010, would have had a difficult time living up to the groundbreaking status of version 3.0. With version 3.0, C# had moved the language firmly out from the shadow of Java and into prominence. The language was quickly becoming elegant.

The next version did introduce some interesting new features:

- [Dynamic binding](#)
- [Named/optional arguments](#)
- [Generic covariant and contravariant](#)
- [Embedded interop types](#)

Embedded interop types eased the deployment pain of creating COM interop assemblies for your application. Generic covariance and contravariance give you more power to use generics, but they're a bit academic and probably most appreciated by framework and library authors. Named and optional parameters let you eliminate many method overloads and provide convenience. But none of those features are exactly paradigm altering.

The major feature was the introduction of the `dynamic` keyword. The `dynamic` keyword introduced into C# version 4.0 the ability to override the compiler on compile-time typing. By using the `dynamic` keyword, you can create constructs similar to dynamically typed languages like JavaScript. You can create a `dynamic x = "a string"` and then add six to it, leaving it up to the runtime to sort out what should happen next.

Dynamic binding gives you the potential for errors but also great power within the language.

C# version 5.0

C# version 5.0, released with Visual Studio 2012, was a focused version of the language. Nearly all of the effort for that version went into another groundbreaking language concept: the `async` and `await` model for asynchronous programming. Here's the major features list:

- [Asynchronous members](#)
- [Caller info attributes](#)

See Also

- [Code Project: Caller Info Attributes in C# 5.0](#)

The caller info attribute lets you easily retrieve information about the context in which you're running without resorting to a ton of boilerplate reflection code. It has many uses in diagnostics and logging tasks.

But `async` and `await` are the real stars of this release. When these features came out in 2012, C# changed the game again by baking asynchrony into the language as a first-class participant. If you've ever dealt with long running operations and the implementation of webs of callbacks, you probably loved this language feature.

C# version 6.0

With versions 3.0 and 5.0, C# had added major new features in an object-oriented language. With version 6.0, released with Visual Studio 2015, it would go away from doing a dominant killer feature and instead release many smaller features that made C# programming more productive. Here are some of them:

- [Static imports](#)
- [Exception filters](#)
- [Auto-property initializers](#)
- [Expression bodied members](#)
- [Null propagator](#)
- [String interpolation](#)
- [nameof operator](#)

Other new features include:

- Index initializers
- Await in catch/finally blocks
- Default values for getter-only properties

Each of these features is interesting in its own right. But if you look at them altogether, you see an interesting pattern. In this version, C# eliminated language boilerplate to make code more terse and readable. So for fans of clean, simple code, this language version was a huge win.

They did one other thing along with this version, though it's not a traditional language feature in itself. They released [Roslyn the compiler as a service](#). The C# compiler is now written in C#, and you can use the compiler as part of your programming efforts.

C# version 7.0

C# version 7.0 was released with Visual Studio 2017. This version has some evolutionary and cool stuff in the vein of C# 6.0, but without the compiler as a service. Here are some of the new features:

- Out variables
- [Tuples and deconstruction](#)
- [Pattern matching](#)
- Local functions
- Expanded expression bodied members
- [Ref locals and returns](#)

Other features included:

- [Discards](#)
- Binary Literals and Digit Separators
- [Throw expressions](#)

All of these features offer cool new capabilities for developers and the opportunity to write even cleaner code than ever. A highlight is condensing the declaration of variables to use with the `out` keyword and by allowing multiple return values via tuple.

But C# is being put to ever broader use. .NET Core now targets any operating system and has its eyes firmly on the cloud and on portability. These new capabilities certainly occupy the language designers' thoughts and time, in addition to coming up with new features.

C# version 7.1

C# started releasing *point releases* with C# 7.1. This version added the [language version selection](#) configuration element, three new language features, and new compiler behavior.

The new language features in this release are:

- [async](#) [Main](#) [method](#)
 - The entry point for an application can have the `async` modifier.
- [default](#) [literal expressions](#)
 - You can use default literal expressions in default value expressions when the target type can be inferred.
- Inferred tuple element names
 - The names of tuple elements can be inferred from tuple initialization in many cases.
- Pattern matching on generic type parameters
 - You can use pattern match expressions on variables whose type is a generic type parameter.

Finally, the compiler has two options `-refout` and `-refonly` that control reference assembly generation

C# version 7.2

C# 7.2 added several small language features:

- Initializers on `stackalloc` arrays.
- Use `fixed` statements with any type that supports a pattern.
- Access fixed fields without pinning.
- Reassign `ref` local variables.
- Declare `readonly struct` types, to indicate that a struct is immutable and should be passed as an `in` parameter to its member methods.
- Add the `in` modifier on parameters, to specify that an argument is passed by reference but not modified by the called method.
- Use the `ref readonly` modifier on method returns, to indicate that a method returns its value by reference but doesn't allow writes to that object.
- Declare `ref struct` types, to indicate that a struct type accesses managed memory directly and must always be stack allocated.
- Use additional generic constraints.
- [Non-trailing named arguments](#)
 - Named arguments can be followed by positional arguments.
- Leading underscores in numeric literals
 - Numeric literals can now have leading underscores before any printed digits.
- [private protected](#) [access modifier](#)
 - The `private protected` access modifier enables access for derived classes in the same assembly.
- Conditional `ref` expressions
 - The result of a conditional expression (`?:`) can now be a reference.

C# version 7.3

There are two main themes to the C# 7.3 release. One theme provides features that enable safe code to be as performant as unsafe code. The second theme provides incremental improvements to existing features. New compiler options were also added in this release.

The following new features support the theme of better performance for safe code:

- You can access fixed fields without pinning.
- You can reassign `ref` local variables.
- You can use initializers on `stackalloc` arrays.
- You can use `fixed` statements with any type that supports a pattern.
- You can use more generic constraints.

The following enhancements were made to existing features:

- You can test `==` and `!=` with tuple types.
- You can use expression variables in more locations.
- You may attach attributes to the backing field of auto-implemented properties.
- Method resolution when arguments differ by `in` has been improved.
- Overload resolution now has fewer ambiguous cases.

The new compiler options are:

- `-publicsign` to enable Open Source Software (OSS) signing of assemblies.
- `-pathmap` to provide a mapping for source directories.

C# version 8.0

C# 8.0 is the first major C# release that specifically targets .NET Core. Some features rely on new CLR capabilities, others on library types added only in .NET Core. C# 8.0 adds the following features and enhancements to the C# language:

- [Readonly members](#)
- [Default interface methods](#)
- [Pattern matching enhancements](#):
 - [Switch expressions](#)
 - [Property patterns](#)
 - [Tuple patterns](#)
 - [Positional patterns](#)
- [Using declarations](#)
- [Static local functions](#)
- [Disposable ref structs](#)
- [Nullable reference types](#)
- [Asynchronous streams](#)
- [Indices and ranges](#)
- [Null-coalescing assignment](#)
- [Unmanaged constructed types](#)
- [Stackalloc in nested expressions](#)
- [Enhancement of interpolated verbatim strings](#)

Default interface members require enhancements in the CLR. Those features were added in the CLR for .NET Core 3.0. Ranges and indexes, and asynchronous streams require new types in the .NET Core 3.0 libraries. Nullable reference types, while implemented in the compiler, is much more useful when libraries are annotated to provide semantic information regarding the null state of arguments and return values. Those annotations are being added in the .NET Core libraries.

C# version 9

C# 9 was released with .NET 5. It's the default language version for any assembly that targets the .NET 5 release. It contains the following new and enhanced features:

- [Records](#)
- [Init only setters](#)
- [Top-level statements](#)
- [Pattern matching enhancements](#)
- [Performance and interop](#)
 - [Native sized integers](#)
 - [Function pointers](#)
 - [Suppress emitting localsinit flag](#)
- [Fit and finish features](#)
 - [Target-typed `new` expressions](#)
 - [static anonymous functions](#)
 - [Target-typed conditional expressions](#)
 - [Covariant return types](#)
 - [Extension `GetEnumerator` support for `foreach` loops](#)
 - [Lambda discard parameters](#)
 - [Attributes on local functions](#)
- [Support for code generators](#)
 - [Module initializers](#)
 - [New features for partial methods](#)

C# 9 continues three of the themes from previous releases: removing ceremony, separating data from algorithms, and providing more patterns in more places.

[Top level statements](#) means your main program is simpler to read. There's less need for ceremony: a namespace, a `Program` class, and `static void Main()` are all unnecessary.

The introduction of [records](#) provide a concise syntax for reference types that follow value semantics for equality. You'll use these types to define data containers that typically define minimal behavior. [Init-only setters](#) provide the capability for non-destructive mutation (`with` expressions) in records. C# 9 also adds [covariant return types](#) so that derived records can override virtual methods and return a type derived from the base method's return type.

The [pattern matching](#) capabilities have been expanded in several ways. Numeric types now support *range patterns*. Patterns can be combined using `and`, `or`, and `not` patterns. Parentheses can be added to clarify more complex patterns.

Another set of features supports high-performance computing in C#:

- The `nint` and `nuint` types model the native-size integer types on the target CPU.
- [Function pointers](#) provide delegate-like functionality while avoiding the allocations necessary to create a delegate object.
- The `localsinit` instruction can be omitted to save instructions.

Another set of improvements supports scenarios where *code generators* add functionality:

- [Module initializers](#) are methods that the runtime calls when an assembly loads.
- [Partial methods](#) support new accessibility modifiers and non-void return types. In those cases, an implementation must be provided.

C# 9 adds many other small features that improve developer productivity, both writing and reading code:

- Target-type `new` expressions
- `static` anonymous functions
- Target-type conditional expressions
- Extension `GetEnumerator()` support for `foreach` loops
- Lambda expressions can declare discard parameters
- Attributes can be applied to local functions

The C# 9 release continues the work to keep C# a modern, general-purpose programming language. Features continue to support modern workloads and application types.

Article [originally published on the NDepend blog](#), courtesy of Erik Dietrich and Patrick Smacchia.

Relationships between language features and library types

12/28/2021 • 2 minutes to read • [Edit Online](#)

The C# language definition requires a standard library to have certain types and certain accessible members on those types. The compiler generates code that uses these required types and members for many different language features. When necessary, there are NuGet packages that contain types needed for newer versions of the language when writing code for environments where those types or members have not been deployed yet.

This dependency on standard library functionality has been part of the C# language since its first version. In that version, examples included:

- [Exception](#) - used for all compiler generated exceptions.
- [String](#) - the C# `string` type is a synonym for [String](#).
- [Int32](#) - synonym of `int`.

That first version was simple: the compiler and the standard library shipped together, and there was only one version of each.

Subsequent versions of C# have occasionally added new types or members to the dependencies. Examples include: [INotifyCompletion](#), [CallerFilePathAttribute](#) and [CallerMemberNameAttribute](#). C# 7.0 continues this by adding a dependency on [ValueTuple](#) to implement the [tuples](#) language feature.

The language design team works to minimize the surface area of the types and members required in a compliant standard library. That goal is balanced against a clean design where new library features are incorporated seamlessly into the language. There will be new features in future versions of C# that require new types and members in a standard library. It's important to understand how to manage those dependencies in your work.

Managing your dependencies

C# compiler tools are now decoupled from the release cycle of the .NET libraries on supported platforms. In fact, different .NET libraries have different release cycles: the .NET Framework on Windows is released as a Windows Update, .NET Core ships on a separate schedule, and the Xamarin versions of library updates ship with the Xamarin tools for each target platform.

The majority of time, you won't notice these changes. However, when you are working with a newer version of the language that requires features not yet in the .NET libraries on that platform, you'll reference the NuGet packages to provide those new types. As the platforms your app supports are updated with new framework installations, you can remove the extra reference.

This separation means you can use new language features even when you are targeting machines that may not have the corresponding framework.

Version and update considerations for C# developers

12/28/2021 • 2 minutes to read • [Edit Online](#)

Compatibility is a very important goal as new features are added to the C# language. In almost all cases, existing code can be recompiled with a new compiler version without any issue.

More care may be required when you adopt new language features in a library. You may be creating a new library with features found in the latest version and need to ensure apps built using previous versions of the compiler can use it. Or you may be upgrading an existing library and many of your users may not have upgraded versions yet. As you make decisions on adopting new features, you'll need to consider two variations of compatibility: source-compatible and binary-compatible.

Binary-compatible changes

Changes to your library are **binary-compatible** when your updated library can be used without rebuilding applications and libraries that use it. Dependent assemblies are not required to be rebuilt, nor are any source code changes required.

Source-compatible changes

Changes to your library are **source-compatible** when applications and libraries that use your library do not require source code changes, but the source must be recompiled against the new version to work correctly.

Incompatible changes

If a change is neither **source-compatible** nor **binary-compatible**, source code changes along with recompilation are required in dependent libraries and applications.

Evaluate your library

These compatibility concepts affect the public and protected declarations for your library, not its internal implementation. Adopting any new features internally are always **binary-compatible**.

binary-compatible changes provide new syntax that generates the same compiled code for public declarations as the older syntax. For example, changing a method to an expression-bodied member is a **binary-compatible** change:

Original code:

```
public double CalculateSquare(double value)
{
    return value * value;
}
```

New code:

```
public double CalculateSquare(double value) => value * value;
```

source-compatible changes introduce syntax that changes the compiled code for a public member, but in a

way that is compatible with existing call sites. For example, changing a method signature from a by value parameter to an `in` by reference parameter is source-compatible, but not binary-compatible:

Original code:

```
public double CalculateSquare(double value) => value * value;
```

New code:

```
public double CalculateSquare(in double value) => value * value;
```

The [What's new](#) articles note if introducing a feature that affects public declarations is source-compatible or binary-compatible.

Create record types

12/28/2021 • 12 minutes to read • [Edit Online](#)

C# 9 introduces *records*, a new reference type that you can create instead of classes or structs. C# 10 adds *record structs* so that you can define records as value types. Records are distinct from classes in that record types use *value-based equality*. Two variables of a record type are equal if the record type definitions are identical, and if for every field, the values in both records are equal. Two variables of a class type are equal if the objects referred to are the same class type and the variables refer to the same object. Value-based equality implies other capabilities you'll probably want in record types. The compiler generates many of those members when you declare a `record` instead of a `class`. The compiler generates those same methods for `record struct` types.

In this tutorial, you'll learn how to:

- Decide if you should declare a `class` or a `record`.
- Declare record types and positional record types.
- Substitute your methods for compiler generated methods in records.

Prerequisites

You'll need to set up your machine to run .NET 6 or later, including the C# 10 or later compiler. The C# 10 compiler is available starting with [Visual Studio 2022](#) or the [.NET 6 SDK](#).

Characteristics of records

You define a *record* by declaring a type with the `record` keyword, instead of the `class` or `struct` keyword. Optionally, you can declare a `record class` to clarify that it's a reference type. A record is a reference type and follows value-based equality semantics. You can define a `record struct` to create a record that is a value type. To enforce value semantics, the compiler generates several methods for your record type (both for `record class` types and `record struct` types):

- An override of [Object.Equals\(Object\)](#).
- A virtual `Equals` method whose parameter is the record type.
- An override of [Object.GetHashCode\(\)](#).
- Methods for `operator ==` and `operator !=`.
- Record types implement [System.IEquatable<T>](#).

Records also provide an override of [Object.ToString\(\)](#). The compiler synthesizes methods for displaying records using [Object.ToString\(\)](#). You'll explore those members as you write the code for this tutorial. Records support `with` expressions to enable non-destructive mutation of records.

You can also declare *positional records* using a more concise syntax. The compiler synthesizes more methods for you when you declare positional records:

- A primary constructor whose parameters match the positional parameters on the record declaration.
- Public properties for each parameter of a primary constructor. These properties are *init-only* for `record class` types and `readonly record struct` types. For `record struct` types, they're *read-write*.
- A `Deconstruct` method to extract properties from the record.

Build temperature data

Data and statistics are among the scenarios where you'll want to use records. For this tutorial, you'll build an application that computes *degree days* for different uses. *Degree days* are a measure of heat (or lack of heat) over a period of days, weeks, or months. Degree days track and predict energy usage. More hotter days means more air conditioning, and more colder days means more furnace usage. Degree days help manage plant populations and correlate to plant growth as the seasons change. Degree days help track animal migrations for species that travel to match climate.

The formula is based on the mean temperature on a given day and a baseline temperature. To compute degree days over time, you'll need the high and low temperature each day for a period of time. Let's start by creating a new application. Make a new console application. Create a new record type in a new file named "DailyTemperature.cs":

```
public readonly record struct DailyTemperature(double HighTemp, double LowTemp);
```

The preceding code defines a *positional record*. The `DailyTemperature` record is a `readonly record struct`, because you don't intend to inherit from it, and it should be immutable. The `HighTemp` and `LowTemp` properties are *init only properties*, meaning they can be set in the constructor or using a property initializer. If you wanted the positional parameters to be read-write, you declare a `record struct` instead of a `readonly record struct`. The `DailyTemperature` type also has a *primary constructor* that has two parameters that match the two properties. You use the primary constructor to initialize a `DailyTemperature` record. The following code creates and initializes several `DailyTemperature` records. The first uses named parameters to clarify the `HighTemp` and `LowTemp`. The remaining initializers use positional parameters to initialize the `HighTemp` and `LowTemp`:

```
private static DailyTemperature[] data = new DailyTemperature[]
{
    new DailyTemperature(HighTemp: 57, LowTemp: 30),
    new DailyTemperature(60, 35),
    new DailyTemperature(63, 33),
    new DailyTemperature(68, 29),
    new DailyTemperature(72, 47),
    new DailyTemperature(75, 55),
    new DailyTemperature(77, 55),
    new DailyTemperature(72, 58),
    new DailyTemperature(70, 47),
    new DailyTemperature(77, 59),
    new DailyTemperature(85, 65),
    new DailyTemperature(87, 65),
    new DailyTemperature(85, 72),
    new DailyTemperature(83, 68),
    new DailyTemperature(77, 65),
    new DailyTemperature(72, 58),
    new DailyTemperature(77, 55),
    new DailyTemperature(76, 53),
    new DailyTemperature(80, 60),
    new DailyTemperature(85, 66)
};
```

You can add your own properties or methods to records, including positional records. You'll need to compute the mean temperature for each day. You can add that property to the `DailyTemperature` record:

```
public readonly record struct DailyTemperature(double HighTemp, double LowTemp)
{
    public double Mean => (HighTemp + LowTemp) / 2.0;
}
```

Let's make sure you can use this data. Add the following code to your `Main` method:

```
foreach (var item in data)
    Console.WriteLine(item);
```

Run your application, and you'll see output that looks similar to the following display (several rows removed for space):

```
DailyTemperature { HighTemp = 57, LowTemp = 30, Mean = 43.5 }
DailyTemperature { HighTemp = 60, LowTemp = 35, Mean = 47.5 }

DailyTemperature { HighTemp = 80, LowTemp = 60, Mean = 70 }
DailyTemperature { HighTemp = 85, LowTemp = 66, Mean = 75.5 }
```

The preceding code shows the output from the override of `ToString` synthesized by the compiler. If you prefer different text, you can write your own version of `ToString` that prevents the compiler from synthesizing a version for you.

Compute degree days

To compute degree days, you take the difference from a baseline temperature and the mean temperature on a given day. To measure heat over time, you discard any days where the mean temperature is below the baseline. To measure cold over time, you discard any days where the mean temperature is above the baseline. For example, the U.S. uses 65F as the base for both heating and cooling degree days. That's the temperature where no heating or cooling is needed. If a day has a mean temperature of 70F, that day is five cooling degree days and zero heating degree days. Conversely, if the mean temperature is 55F, that day is 10 heating degree days and 0 cooling degree days.

You can express these formulas as a small hierarchy of record types: an abstract degree day type and two concrete types for heating degree days and cooling degree days. These types can also be positional records. They take a baseline temperature and a sequence of daily temperature records as arguments to the primary constructor:

```
public abstract record DegreeDays(double BaseTemperature, IEnumerable<DailyTemperature> TempRecords);

public sealed record HeatingDegreeDays(double BaseTemperature, IEnumerable<DailyTemperature> TempRecords)
    : DegreeDays(BaseTemperature, TempRecords)
{
    public double DegreeDays => TempRecords.Where(s => s.Mean < BaseTemperature).Sum(s => BaseTemperature - s.Mean);
}

public sealed record CoolingDegreeDays(double BaseTemperature, IEnumerable<DailyTemperature> TempRecords)
    : DegreeDays(BaseTemperature, TempRecords)
{
    public double DegreeDays => TempRecords.Where(s => s.Mean > BaseTemperature).Sum(s => s.Mean - BaseTemperature);
}
```

The abstract `DegreeDays` record is the shared base class for both the `HeatingDegreeDays` and `CoolingDegreeDays` records. The primary constructor declarations on the derived records show how to manage base record initialization. Your derived record declares parameters for all the parameters in the base record primary constructor. The base record declares and initializes those properties. The derived record doesn't hide them, but only creates and initializes properties for parameters that aren't declared in its base record. In this example, the derived records don't add new primary constructor parameters. Test your code by adding the following code to

your `Main` method:

```
var heatingDegreeDays = new HeatingDegreeDays(65, data);
Console.WriteLine(heatingDegreeDays);

var coolingDegreeDays = new CoolingDegreeDays(65, data);
Console.WriteLine(coolingDegreeDays);
```

You'll get output like the following display:

```
HeatingDegreeDays { BaseTemperature = 65, TempRecords = record_types.DailyTemperature[], DegreeDays = 85 }
CoolingDegreeDays { BaseTemperature = 65, TempRecords = record_types.DailyTemperature[], DegreeDays = 71.5 }
```

Define compiler-synthesized methods

Your code calculates the correct number of heating and cooling degree days over that period of time. But this example shows why you may want to replace some of the synthesized methods for records. You can declare your own version of any of the compiler-synthesized methods in a record type except the clone method. The clone method has a compiler-generated name and you can't provide a different implementation. These synthesized methods include a copy constructor, the members of the `System.IEquatable<T>` interface, equality and inequality tests, and `GetHashCode()`. For this purpose, you'll synthesize `PrintMembers`. You could also declare your own `ToString`, but `PrintMembers` provides a better option for inheritance scenarios. To provide your own version of a synthesized method, the signature must match the synthesized method.

The `TempRecords` element in the console output isn't useful. It displays the type, but nothing else. You can change this behavior by providing your own implementation of the synthesized `PrintMembers` method. The signature depends on modifiers applied to the `record` declaration:

- If a record type is `sealed`, or a `record struct`, the signature is
`private bool PrintMembers(StringBuilder builder);`
- If a record type isn't `sealed` and derives from `object` (that is, it doesn't declare a base record), the signature is
`protected virtual bool PrintMembers(StringBuilder builder);`
- If a record type isn't `sealed` and derives from another record, the signature is
`protected override bool PrintMembers(StringBuilder builder);`

These rules are easiest to comprehend through understanding the purpose of `PrintMembers`. `PrintMembers` adds information about each property in a record type to a string. The contract requires base records to add their members to the display and assumes derived members will add their members. Each record type synthesizes a `ToString` override that looks similar to the following example for `HeatingDegreeDays`:

```
public override string ToString()
{
    StringBuilder stringBuilder = new StringBuilder();
    stringBuilder.Append("HeatingDegreeDays");
    stringBuilder.Append(" { ");
    if (PrintMembers(stringBuilder))
    {
        stringBuilder.Append(" ");
    }
    stringBuilder.Append("}");
    return stringBuilder.ToString();
}
```

You declare a `PrintMembers` method in the `DegreeDays` record that doesn't print the type of the collection:


```
protected virtual bool PrintMembers(StringBuilder stringBuilder)
{
    stringBuilder.Append($"BaseTemperature = {BaseTemperature}");
    return true;
}
```

The signature declares a `virtual protected` method to match the compiler's version. Don't worry if you get the accessors wrong; the language enforces the correct signature. If you forget the correct modifiers for any synthesized method, the compiler issues warnings or errors that help you get the right signature.

In C# 10 and later, you can declare the `ToString` method as `sealed` in a record type. That prevents derived records from providing a new implementation. Derived records will still contain the `PrintMembers` override. You would do seal `ToString` if you didn't want it to display the runtime type of the record. In the preceding example, you'd lose the information on where the record was measuring heating or cooling degree days.

Non-destructive mutation

The synthesized members in a positional record class don't modify the state of the record. The goal is that you can more easily create immutable records. Remember that you declare a `readonly record struct` to create an immutable record struct. Look again at the preceding declarations for `HeatingDegreeDays` and `CoolingDegreeDays`. The members added perform computations on the values for the record, but don't mutate state. Positional records make it easier for you to create immutable reference types.

Creating immutable reference types means you'll want to use non-destructive mutation. You create new record instances that are similar to existing record instances using `with` expressions. These expressions are a copy construction with additional assignments that modify the copy. The result is a new record instance where each property has been copied from the existing record and optionally modified. The original record is unchanged.

Let's add a couple features to your program that demonstrate `with` expressions. First, let's create a new record to compute growing degree days using the same data. *Growing degree days* typically uses 41F as the baseline and measures temperatures above the baseline. To use the same data, you can create a new record that is similar to the `coolingDegreeDays`, but with a different base temperature:

```
// Growing degree days measure warming to determine plant growing rates
var growingDegreeDays = coolingDegreeDays with { BaseTemperature = 41 };
Console.WriteLine(growingDegreeDays);
```

You can compare the number of degrees computed to the numbers generated with a higher baseline temperature. Remember that records are *reference types* and these copies are shallow copies. The array for the data isn't copied, but both records refer to the same data. That fact is an advantage in one other scenario. For growing degree days, it's useful to keep track of the total for the previous five days. You can create new records with different source data using `with` expressions. The following code builds a collection of these accumulations, then displays the values:

```
// showing moving accumulation of 5 days using range syntax
List<CoolingDegreeDays> movingAccumulation = new();
int rangeSize = (data.Length > 5) ? 5 : data.Length;
for (int start = 0; start < data.Length - rangeSize; start++)
{
    var fiveDayTotal = growingDegreeDays with { TempRecords = data[start..(start + rangeSize)] };
    movingAccumulation.Add(fiveDayTotal);
}
Console.WriteLine();
Console.WriteLine("Total degree days in the last five days");
foreach(var item in movingAccumulation)
{
    Console.WriteLine(item);
}
```

You can also use `with` expressions to create copies of records. Don't specify any properties between the braces for the `with` expression. That means create a copy, and don't change any properties:

```
var growingDegreeDaysCopy = growingDegreeDays with { };
```

Run the finished application to see the results.

Summary

This tutorial showed several aspects of records. Records provide concise syntax for types where the fundamental use is storing data. For object-oriented classes, the fundamental use is defining responsibilities. This tutorial focused on *positional records*, where you can use a concise syntax to declare the properties for a record. The compiler synthesizes several members of the record for copying and comparing records. You can add any other members you need for your record types. You can create immutable record types knowing that none of the compiler-generated members would mutate state. And `with` expressions make it easy to support non-destructive mutation.

Records add another way to define types. You use `class` definitions to create object-oriented hierarchies that focus on the responsibilities and behavior of objects. You create `struct` types for data structures that store data and are small enough to copy efficiently. You create `record` types when you want value-based equality and comparison, don't want to copy values, and want to use reference variables. You create `record struct` types when you want the features of records for a type that is small enough to copy efficiently.

You can learn more about records in the [C# language reference article for the record type](#) and the [proposed record type specification](#) and [record struct specification](#).

Tutorial: Explore ideas using top-level statements to build code as you learn

12/28/2021 • 8 minutes to read • [Edit Online](#)

In this tutorial, you'll learn how to:

- Learn the rules governing your use of top-level statements.
- Use top-level statements to explore algorithms.
- Refactor explorations into reusable components.

Prerequisites

You'll need to set up your machine to run .NET 6, which includes the C# 10 compiler. The C# 10 compiler is available starting with [Visual Studio 2022](#) or [.NET 6 SDK](#).

This tutorial assumes you're familiar with C# and .NET, including either Visual Studio or the .NET CLI.

Start exploring

Top-level statements enable you to avoid the extra ceremony required by placing your program's entry point in a static method in a class. The typical starting point for a new console application looks like the following code:

```
using System;

namespace Application
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Hello World!");
        }
    }
}
```

The preceding code is the result of running the `dotnet new console` command and creating a new console application. Those 11 lines contain only one line of executable code. You can simplify that program with the new top-level statements feature. That enables you to remove all but two of the lines in this program:

```
// See https://aka.ms/new-console-template for more information
Console.WriteLine("Hello, World!");
```

IMPORTANT

The C# templates for .NET 6 use *top level statements*. Your application may not match the code in this article, if you've already upgraded to the .NET 6 previews. For more information see the article on [New C# templates generate top level statements](#)

The .NET 6 SDK also adds a set of *implicit* `global using` directives for projects that use the following SDKs:

- Microsoft.NET.Sdk
- Microsoft.NET.Sdk.Web
- Microsoft.NET.Sdk.Worker

These implicit `global using` directives include the most common namespaces for the project type.

This feature simplifies what's needed to begin exploring new ideas. You can use top-level statements for scripting scenarios, or to explore. Once you've got the basics working, you can start refactoring the code and create methods, classes, or other assemblies for reusable components you've built. Top-level statements do enable quick experimentation and beginner tutorials. They also provide a smooth path from experimentation to full programs.

Top-level statements are executed in the order they appear in the file. Top-level statements can only be used in one source file in your application. The compiler generates an error if you use them in more than one file.

Build a magic .NET answer machine

For this tutorial, let's build a console application that answers a "yes" or "no" question with a random answer. You'll build out the functionality step by step. You can focus on your task rather than ceremony needed for the structure of a typical program. Then, once you're happy with the functionality, you can refactor the application as you see fit.

A good starting point is to write the question back to the console. You can start by writing the following code:

```
Console.WriteLine(args);
```

You don't declare an `args` variable. For the single source file that contains your top-level statements, the compiler recognizes `args` to mean the command-line arguments. The type of `args` is a `string[]`, as in all C# programs.

You can test your code by running the following `dotnet run` command:

```
dotnet run -- Should I use top level statements in all my programs?
```

The arguments after the `--` on the command line are passed to the program. You can see the type of the `args` variable, because that's what's printed to the console:

```
System.String[]
```

To write the question to the console, you'll need to enumerate the arguments and separate them with a space. Replace the `WriteLine` call with the following code:

```

Console.WriteLine();
foreach(var s in args)
{
    Console.Write(s);
    Console.Write(' ');
}
Console.WriteLine();

```

Now, when you run the program, it will correctly display the question as a string of arguments.

Respond with a random answer

After echoing the question, you can add the code to generate the random answer. Start by adding an array of possible answers:

```

string[] answers =
{
    "It is certain.",      "Reply hazy, try again.",    "Don't count on it.",
    "It is decidedly so.", "Ask again later.",         "My reply is no.",
    "Without a doubt.",   "Better not tell you now.",  "My sources say no.",
    "Yes - definitely.",  "Cannot predict now.",      "Outlook not so good.",
    "You may rely on it.", "Concentrate and ask again.", "Very doubtful.",
    "As I see it, yes.",
    "Most likely.",
    "Outlook good.",
    "Yes.",
    "Signs point to yes.",
};

```

This array has ten answers that are affirmative, five that are non-committal, and five that are negative. Next, add the following code to generate and display a random answer from the array:

```

var index = new Random().Next(answers.Length - 1);
Console.WriteLine(answers[index]);

```

You can run the application again to see the results. You should see something like the following output:

```

dotnet run -- Should I use top level statements in all my programs?

Should I use top level statements in all my programs?
Better not tell you now.

```

This code answers the questions, but let's add one more feature. You'd like your question app to simulate thinking about the answer. You can do that by adding a bit of ASCII animation, and pausing while working. Add the following code after the line that echoes the question:

```

for (int i = 0; i < 20; i++)
{
    Console.Write("| -");
    await Task.Delay(50);
    Console.Write("\b\b\b");
    Console.Write("/ \\");
    await Task.Delay(50);
    Console.Write("\b\b\b");
    Console.Write("- |");
    await Task.Delay(50);
    Console.Write("\b\b\b");
    Console.Write("\\ /");
    await Task.Delay(50);
    Console.Write("\b\b\b");
}
Console.WriteLine();

```

You'll also need to add a `using` statement to the top of the source file:

```
using System.Threading.Tasks;
```

The `using` statements must be before any other statements in the file. Otherwise, it's a compiler error. You can run the program again and see the animation. That makes a better experience. Experiment with the length of the delay to match your taste.

The preceding code creates a set of spinning lines separated by a space. Adding the `await` keyword instructs the compiler to generate the program entry point as a method that has the `async` modifier, and returns a [System.Threading.Tasks.Task](#). This program doesn't return a value, so the program entry point returns a `Task`. If your program returns an integer value, you would add a return statement to the end of your top-level statements. That return statement would specify the integer value to return. If your top-level statements include an `await` expression, the return type becomes [System.Threading.Tasks.Task<TResult>](#).

Refactoring for the future

Your program should look like the following code:

```

Console.WriteLine();
foreach(var s in args)
{
    Console.Write(s);
    Console.Write(' ');
}
Console.WriteLine();

for (int i = 0; i < 20; i++)
{
    Console.Write("| -");
    await Task.Delay(50);
    Console.Write("\b\b\b\b");
    Console.Write("/ \\\");
    await Task.Delay(50);
    Console.Write("\b\b\b\b");
    Console.Write("- |");
    await Task.Delay(50);
    Console.Write("\b\b\b\b");
    Console.Write("\\ /");
    await Task.Delay(50);
    Console.Write("\b\b\b\b");
}
Console.WriteLine();

string[] answers =
{
    "It is certain.",      "Reply hazy, try again.",    "Don't count on it.",
    "It is decidedly so.", "Ask again later.",          "My reply is no.",
    "Without a doubt.",    "Better not tell you now.",  "My sources say no.",
    "Yes - definitely.",   "Cannot predict now.",       "Outlook not so good.",
    "You may rely on it.", "Concentrate and ask again.", "Very doubtful.",
    "As I see it, yes.",
    "Most likely.",
    "Outlook good.",
    "Yes.",
    "Signs point to yes.",
};

var index = new Random().Next(answers.Length - 1);
Console.WriteLine(answers[index]);

```

The preceding code is reasonable. It works. But it isn't reusable. Now that you have the application working, it's time to pull out reusable parts.

One candidate is the code that displays the waiting animation. That snippet can become a method:

You can start by creating a local function in your file. Replace the current animation with the following code:

```

await ShowConsoleAnimation();

static async Task ShowConsoleAnimation()
{
    for (int i = 0; i < 20; i++)
    {
        Console.Write("| -");
        await Task.Delay(50);
        Console.Write("\b\b\b");
        Console.Write("/ \");
        await Task.Delay(50);
        Console.Write("\b\b\b");
        Console.Write("- |");
        await Task.Delay(50);
        Console.Write("\b\b\b");
        Console.Write("\ \ /");
        await Task.Delay(50);
        Console.Write("\b\b\b");
    }
    Console.WriteLine();
}

```

The preceding code creates a local function inside your main method. That's still not reusable. So, extract that code into a class. Create a new file named *utilities.cs* and add the following code:

```

namespace MyNamespace
{
    public static class Utilities
    {
        public static async Task ShowConsoleAnimation()
        {
            for (int i = 0; i < 20; i++)
            {
                Console.Write("| -");
                await Task.Delay(50);
                Console.Write("\b\b\b");
                Console.Write("/ \");
                await Task.Delay(50);
                Console.Write("\b\b\b");
                Console.Write("- |");
                await Task.Delay(50);
                Console.Write("\b\b\b");
                Console.Write("\ \ /");
                await Task.Delay(50);
                Console.Write("\b\b\b");
            }
            Console.WriteLine();
        }
    }
}

```

A file that has top-level statements can also contain namespaces and types at the end of the file, after the top-level statements. But for this tutorial you put the animation method in a separate file to make it more readily reusable.

Finally, you can clean the animation code to remove some duplication:


```
foreach (string s in new[] { "| -", "/ \\", "- |", "\\ /", })
{
    Console.Write(s);
    await Task.Delay(50);
    Console.Write("\b\b\b");
}
```

Now you have a complete application, and you've refactored the reusable parts for later use. You can call the new utility method from your top-level statements, as shown below in the finished version of the main program:

```
using MyNamespace;

Console.WriteLine();
foreach(var s in args)
{
    Console.Write(s);
    Console.Write(' ');
}
Console.WriteLine();

await Utilities.ShowConsoleAnimation();

string[] answers =
{
    "It is certain.",      "Reply hazy, try again.",    "Don't count on it.",
    "It is decidedly so.", "Ask again later.",          "My reply is no.",
    "Without a doubt.",   "Better not tell you now.",  "My sources say no.",
    "Yes – definitely.",  "Cannot predict now.",       "Outlook not so good.",
    "You may rely on it.", "Concentrate and ask again.", "Very doubtful.",
    "As I see it, yes.",
    "Most likely.",
    "Outlook good.",
    "Yes.",
    "Signs point to yes.",
};

var index = new Random().Next(answers.Length - 1);
Console.WriteLine(answers[index]);
```

The preceding example adds the call to `Utilities.ShowConsoleAnimation`, and adds an additional `using` statement.

Summary

Top-level statements make it easier to create simple programs for use to explore new algorithms. You can experiment with algorithms by trying different snippets of code. Once you've learned what works, you can refactor the code to be more maintainable.

Top-level statements simplify programs that are based on console applications. These include Azure functions, GitHub actions, and other small utilities. For more information, see [Top-level statements \(C# Programming Guide\)](#).

Use pattern matching to build your class behavior for better code

12/28/2021 • 10 minutes to read • [Edit Online](#)

The pattern matching features in C# provide syntax to express your algorithms. You can use these techniques to implement the behavior in your classes. You can combine object-oriented class design with a data-oriented implementation to provide concise code while modeling real-world objects.

In this tutorial, you'll learn how to:

- Express your object oriented classes using data patterns.
- Implement those patterns using C#'s pattern matching features.
- Leverage compiler diagnostics to validate your implementation.

Prerequisites

You'll need to set up your machine to run .NET 5, including the C# 9 compiler. The C# 9 compiler is available starting with [Visual Studio 2019 version 16.8](#) or the [.NET 5 SDK](#).

Build a simulation of a canal lock

In this tutorial, you'll build a C# class that simulates a [canal lock](#). Briefly, a canal lock is a device that raises and lowers boats as they travel between two stretches of water at different levels. A lock has two gates and some mechanism to change the water level.

In its normal operation, a boat enters one of the gates while the water level in the lock matches the water level on the side the boat enters. Once in the lock, the water level is changed to match the water level where the boat will leave the lock. Once the water level matches that side, the gate on the exit side opens. Safety measures make sure an operator can't create a dangerous situation in the canal. The water level can be changed only when both gates are closed. At most one gate can be open. To open a gate, the water level in the lock must match the water level outside the gate being opened.

You can build a C# class to model this behavior. A `CanalLock` class would support commands to open or close either gate. It would have other commands to raise or lower the water. The class should also support properties to read the current state of both gates and the water level. Your methods implement the safety measures.

Define a class

You'll build a console application to test your `CanalLock` class. Create a new console project for .NET 5 using either Visual Studio or the .NET CLI. Then, add a new class and name it `CanalLock`. Next, design your public API, but leave the methods not implemented:

```

public enum WaterLevel
{
    Low,
    High
}
public class CanalLock
{
    // Query canal lock state:
    public WaterLevel CanalLockWaterLevel { get; private set; } = WaterLevel.Low;
    public bool HighWaterGateOpen { get; private set; } = false;
    public bool LowWaterGateOpen { get; private set; } = false;

    // Change the upper gate.
    public void SetHighGate(bool open)
    {
        throw new NotImplementedException();
    }

    // Change the lower gate.
    public void SetLowGate(bool open)
    {
        throw new NotImplementedException();
    }

    // Change water level.
    public void SetWaterLevel(WaterLevel newLevel)
    {
        throw new NotImplementedException();
    }

    public override string ToString() =>
        $"The lower gate is {(LowWaterGateOpen ? "Open" : "Closed")}. " +
        $"The upper gate is {(HighWaterGateOpen ? "Open" : "Closed")}. " +
        $"The water level is {CanalLockWaterLevel}.";
}

```

The preceding code initializes the object so both gates are closed, and the water level is low. Next, write the following test code in your `Main` method to guide you as you create a first implementation of the class:

```

// Create a new canal lock:
var canalGate = new CanalLock();

// State should be doors closed, water level low:
Console.WriteLine(canalGate);

canalGate.SetLowGate(open: true);
Console.WriteLine($"Open the lower gate: {canalGate}");

Console.WriteLine("Boat enters lock from lower gate");

canalGate.SetLowGate(open: false);
Console.WriteLine($"Close the lower gate: {canalGate}");

canalGate.SetWaterLevel(WaterLevel.High);
Console.WriteLine($"Raise the water level: {canalGate}");
Console.WriteLine(canalGate);

canalGate.SetHighGate(open: true);
Console.WriteLine($"Open the higher gate: {canalGate}");

Console.WriteLine("Boat exits lock at upper gate");
Console.WriteLine("Boat enters lock from upper gate");

canalGate.SetHighGate(open: false);
Console.WriteLine($"Close the higher gate: {canalGate}");

canalGate.SetWaterLevel(WaterLevel.Low);
Console.WriteLine($"Lower the water level: {canalGate}");

canalGate.SetLowGate(open: true);
Console.WriteLine($"Open the lower gate: {canalGate}");

Console.WriteLine("Boat exits lock at upper gate");

canalGate.SetLowGate(open: false);
Console.WriteLine($"Close the lower gate: {canalGate}");

```

Next, add a first implementation of each method in the `CanalLock` class. The following code implements the methods of the class without concern to the safety rules. You'll add safety tests later:

```

// Change the upper gate.
public void SetHighGate(bool open)
{
    HighWaterGateOpen = open;
}

// Change the lower gate.
public void SetLowGate(bool open)
{
    LowWaterGateOpen = open;
}

// Change water level.
public void SetWaterLevel(WaterLevel newLevel)
{
    CanalLockWaterLevel = newLevel;
}

```

The tests you've written so far pass. You've implemented the basics. Now, write a test for the first failure condition. At the end of the previous tests, both gates are closed, and the water level is set to low. Add a test to try opening the upper gate:

```

Console.WriteLine("=====");
Console.WriteLine("    Test invalid commands");
// Open "wrong" gate (2 tests)
try
{
    canalGate = new CanalLock();
    canalGate.SetHighGate(open: true);
}
catch (InvalidOperationException)
{
    Console.WriteLine("Invalid operation: Can't open the high gate. Water is low.");
}
Console.WriteLine($"Try to open upper gate: {canalGate}");

```

This test fails because the gate opens. As a first implementation, you could fix it with the following code:

```

// Change the upper gate.
public void SetHighGate(bool open)
{
    if (open && (CanalLockWaterLevel == WaterLevel.High))
        HighWaterGateOpen = true;
    else if (open && (CanalLockWaterLevel == WaterLevel.Low))
        throw new InvalidOperationException("Cannot open high gate when the water is low");
}

```

Your tests pass. But, as you add more tests, you'll add more and more `if` clauses and test different properties. Soon, these methods will get too complicated as you add more conditionals.

Implement the commands with patterns

A better way is to use *patterns* to determine if the object is in a valid state to execute a command. You can express if a command is allowed as a function of three variables: the state of the gate, the level of the water, and the new setting:

NEW SETTING	GATE STATE	WATER LEVEL	RESULT
Closed	Closed	High	Closed
Closed	Closed	Low	Closed
Closed	Open	High	Open
Closed	Open	Low	Closed
Open	Closed	High	Open
Open	Closed	Low	Closed (Error)
Open	Open	High	Open
Open	Open	Low	Closed (Error)

The fourth and last rows in the table have strike through text because they're invalid. The code you're adding now should make sure the high water gate is never opened when the water is low. Those states can be coded as a single switch expression (remember that `false` indicates "Closed"):

```

HighWaterGateOpen = (open, HighWaterGateOpen, CanalLockWaterLevel) switch
{
    (false, false, WaterLevel.High) => false,
    (false, false, WaterLevel.Low) => false,
    (false, true, WaterLevel.High) => false,
    (false, true, WaterLevel.Low) => false, // should never happen
    (true, false, WaterLevel.High) => true,
    (true, false, WaterLevel.Low) => throw new InvalidOperationException("Cannot open high gate when the
water is low"),
    (true, true, WaterLevel.High) => true,
    (true, true, WaterLevel.Low) => false, // should never happen
};

```

Try this version. Your tests pass, validating the code. The full table shows the possible combinations of inputs and results. That means you and other developers can quickly look at the table and see that you've covered all the possible inputs. Even easier, the compiler can help as well. After you add the previous code, you can see that the compiler generates a warning: *CS8524* indicates the switch expression doesn't cover all possible inputs. The reason for that warning is that one of the inputs is an `enum` type. The compiler interprets "all possible inputs" as all inputs from the underlying type, typically an `int`. This `switch` expression only checks the values declared in the `enum`. To remove the warning, you can add a catch-all discard pattern for the last arm of the expression. This condition throws an exception, because it indicates invalid input:

```

_ => throw new InvalidOperationException("Invalid internal state"),

```

The preceding switch arm must be last in your `switch` expression because it matches all inputs. Experiment by moving it earlier in the order. That causes a compiler error *CS8510* for unreachable code in a pattern. The natural structure of switch expressions enables the compiler to generate errors and warnings for possible mistakes. The compiler "safety net" makes it easier for you to create correct code in fewer iterations, and the freedom to combine switch arms with wildcards. The compiler will issue errors if your combination results in unreachable arms you didn't expect, and warnings if you remove an arm that's needed.

The first change is to combine all the arms where the command is to close the gate; that's always allowed. Add the following code as the first arm in your switch expression:

```

(false, _, _) => false,

```

After you add the previous switch arm, you'll get four compiler errors, one on each of the arms where the command is `false`. Those arms are already covered by the newly added arm. You can safely remove those four lines. You intended this new switch arm to replace those conditions.

Next, you can simplify the four arms where the command is to open the gate. In both cases where the water level is high, the gate can be opened. (In one, it's already open.) One case where the water level is low throws an exception, and the other shouldn't happen. It should be safe to throw the same exception if the water lock is already in an invalid state. You can make the following simplifications for those arms:

```

(true, _, WaterLevel.High) => true,
(true, false, WaterLevel.Low) => throw new InvalidOperationException("Cannot open high gate when the water
is low"),
_ => throw new InvalidOperationException("Invalid internal state"),

```

Run your tests again, and they pass. Here's the final version of the `SetHighGate` method:

```
// Change the upper gate.
public void SetHighGate(bool open)
{
    HighWaterGateOpen = (open, HighWaterGateOpen, CanalLockWaterLevel) switch
    {
        (false, _, _) => false,
        (true, _, WaterLevel.High) => true,
        (true, false, WaterLevel.Low) => throw new InvalidOperationException("Cannot open high gate when
the water is low"),
        _ => throw new InvalidOperationException("Invalid internal state"),
    };
}
```

Implement patterns yourself

Now that you've seen the technique, fill in the `SetLowGate` and `SetWaterLevel` methods yourself. Start by adding the following code to test invalid operations on those methods:

```
Console.WriteLine();
Console.WriteLine();
try
{
    canalGate = new CanalLock();
    canalGate.SetWaterLevel(WaterLevel.High);
    canalGate.SetLowGate(open: true);
}
catch (InvalidOperationException)
{
    Console.WriteLine("invalid operation: Can't open the lower gate. Water is high.");
}
Console.WriteLine($"Try to open lower gate: {canalGate}");
// change water level with gate open (2 tests)
Console.WriteLine();
Console.WriteLine();
try
{
    canalGate = new CanalLock();
    canalGate.SetLowGate(open: true);
    canalGate.SetWaterLevel(WaterLevel.High);
}
catch (InvalidOperationException)
{
    Console.WriteLine("invalid operation: Can't raise water when the lower gate is open.");
}
Console.WriteLine($"Try to raise water with lower gate open: {canalGate}");
Console.WriteLine();
Console.WriteLine();
try
{
    canalGate = new CanalLock();
    canalGate.SetWaterLevel(WaterLevel.High);
    canalGate.SetHighGate(open: true);
    canalGate.SetWaterLevel(WaterLevel.Low);
}
catch (InvalidOperationException)
{
    Console.WriteLine("invalid operation: Can't lower water when the high gate is open.");
}
Console.WriteLine($"Try to lower water with high gate open: {canalGate}");
```

Run your application again. You can see the new tests fail, and the canal lock gets into an invalid state. Try to implement the remaining methods yourself. The method to set the lower gate should be similar to the method to set the upper gate. The method that changes the water level has different checks, but should follow a similar

structure. You may find it helpful to use the same process for the method that sets the water level. Start with all four inputs: The state of both gates, the current state of the water level, and the requested new water level. The switch expression should start with:

```
CanalLockWaterLevel = (newLevel, CanalLockWaterLevel, LowWaterGateOpen, HighWaterGateOpen) switch
{
    // elided
};
```

You'll have 16 total switch arms to fill in. Then, test and simplify.

Did you make methods something like this?

```
// Change the lower gate.
public void SetLowGate(bool open)
{
    LowWaterGateOpen = (open, LowWaterGateOpen, CanalLockWaterLevel) switch
    {
        (false, _, _) => false,
        (true, _, WaterLevel.Low) => true,
        (true, false, WaterLevel.High) => throw new InvalidOperationException("Cannot open high gate when
the water is low"),
        _ => throw new InvalidOperationException("Invalid internal state"),
    };
}

// Change water level.
public void SetWaterLevel(WaterLevel newLevel)
{
    CanalLockWaterLevel = (newLevel, CanalLockWaterLevel, LowWaterGateOpen, HighWaterGateOpen) switch
    {
        (WaterLevel.Low, WaterLevel.Low, true, false) => WaterLevel.Low,
        (WaterLevel.High, WaterLevel.High, false, true) => WaterLevel.High,
        (WaterLevel.Low, _, false, false) => WaterLevel.Low,
        (WaterLevel.High, _, false, false) => WaterLevel.High,
        (WaterLevel.Low, WaterLevel.High, false, true) => throw new InvalidOperationException("Cannot lower
water when the high gate is open"),
        (WaterLevel.High, WaterLevel.Low, true, false) => throw new InvalidOperationException("Cannot raise
water when the low gate is open"),
        _ => throw new InvalidOperationException("Invalid internal state"),
    };
}
```

Your tests should pass, and the canal lock should operate safely.

Summary

In this tutorial, you learned to use pattern matching to check the internal state of an object before applying any changes to that state. You can check combinations of properties. Once you've built tables for any of those transitions, you test your code, then simplify for readability and maintainability. These initial refactorings may suggest further refactorings that validate internal state or manage other API changes. This tutorial combined classes and objects with a more data-oriented, pattern-based approach to implement those classes.

Tutorial: Update interfaces with default interface methods in C# 8.0

12/28/2021 • 6 minutes to read • [Edit Online](#)

Beginning with C# 8.0 on .NET Core 3.0, you can define an implementation when you declare a member of an interface. The most common scenario is to safely add members to an interface already released and used by innumerable clients.

In this tutorial, you'll learn how to:

- Extend interfaces safely by adding methods with implementations.
- Create parameterized implementations to provide greater flexibility.
- Enable implementers to provide a more specific implementation in the form of an override.

Prerequisites

You'll need to set up your machine to run .NET Core, including the C# 8.0 compiler. The C# 8.0 compiler is available starting with [Visual Studio 2019 version 16.3](#) or the [.NET Core 3.0 SDK](#).

Scenario overview

This tutorial starts with version 1 of a customer relationship library. You can get the starter application on our [samples repo on GitHub](#). The company that built this library intended customers with existing applications to adopt their library. They provided minimal interface definitions for users of their library to implement. Here's the interface definition for a customer:

```
public interface ICustomer
{
    IEnumerable<IOrder> PreviousOrders { get; }

    DateTime DateJoined { get; }
    DateTime? LastOrder { get; }
    string Name { get; }
    IDictionary<DateTime, string> Reminders { get; }
}
```

They defined a second interface that represents an order:

```
public interface IOrder
{
    DateTime Purchased { get; }
    decimal Cost { get; }
}
```

From those interfaces, the team could build a library for their users to create a better experience for their customers. Their goal was to create a deeper relationship with existing customers and improve their relationships with new customers.

Now, it's time to upgrade the library for the next release. One of the requested features enables a loyalty discount for customers that have lots of orders. This new loyalty discount gets applied whenever a customer makes an order. The specific discount is a property of each individual customer. Each implementation of

`ICustomer` can set different rules for the loyalty discount.

The most natural way to add this functionality is to enhance the `ICustomer` interface with a method to apply any loyalty discount. This design suggestion caused concern among experienced developers: "Interfaces are immutable once they've been released! This is a breaking change!" C# 8.0 adds *default interface implementations* for upgrading interfaces. The library authors can add new members to the interface and provide a default implementation for those members.

Default interface implementations enable developers to upgrade an interface while still enabling any implementors to override that implementation. Users of the library can accept the default implementation as a non-breaking change. If their business rules are different, they can override.

Upgrade with default interface methods

The team agreed on the most likely default implementation: a loyalty discount for customers.

The upgrade should provide the functionality to set two properties: the number of orders needed to be eligible for the discount, and the percentage of the discount. This makes it a perfect scenario for default interface methods. You can add a method to the `ICustomer` interface, and provide the most likely implementation. All existing, and any new implementations can use the default implementation, or provide their own.

First, add the new method to the interface, including the body of the method:

```
// Version 1:
public decimal ComputeLoyaltyDiscount()
{
    DateTime TwoYearsAgo = DateTime.Now.AddYears(-2);
    if ((DateJoined < TwoYearsAgo) && (PreviousOrders.Count() > 10))
    {
        return 0.10m;
    }
    return 0;
}
```

The library author wrote a first test to check the implementation:

```
SampleCustomer c = new SampleCustomer("customer one", new DateTime(2010, 5, 31))
{
    Reminders =
    {
        { new DateTime(2010, 08, 12), "childs's birthday" },
        { new DateTime(1012, 11, 15), "anniversary" }
    }
};

SampleOrder o = new SampleOrder(new DateTime(2012, 6, 1), 5m);
c.AddOrder(o);

o = new SampleOrder(new DateTime(2103, 7, 4), 25m);
c.AddOrder(o);

// Check the discount:
ICustomer theCustomer = c;
Console.WriteLine($"Current discount: {theCustomer.ComputeLoyaltyDiscount()}");
```

Notice the following portion of the test:

```
// Check the discount:
ICustomer theCustomer = c;
Console.WriteLine($"Current discount: {theCustomer.ComputeLoyaltyDiscount()}");
```

That cast from `SampleCustomer` to `ICustomer` is necessary. The `SampleCustomer` class doesn't need to provide an implementation for `ComputeLoyaltyDiscount`; that's provided by the `ICustomer` interface. However, the `SampleCustomer` class doesn't inherit members from its interfaces. That rule hasn't changed. In order to call any method declared and implemented in the interface, the variable must be the type of the interface, `ICustomer` in this example.

Provide parameterization

That's a good start. But, the default implementation is too restrictive. Many consumers of this system may choose different thresholds for number of purchases, a different length of membership, or a different percentage discount. You can provide a better upgrade experience for more customers by providing a way to set those parameters. Let's add a static method that sets those three parameters controlling the default implementation:

```
// Version 2:
public static void SetLoyaltyThresholds(
    TimeSpan ago,
    int minimumOrders = 10,
    decimal percentageDiscount = 0.10m)
{
    length = ago;
    orderCount = minimumOrders;
    discountPercent = percentageDiscount;
}
private static TimeSpan length = new TimeSpan(365 * 2, 0, 0, 0); // two years
private static int orderCount = 10;
private static decimal discountPercent = 0.10m;

public decimal ComputeLoyaltyDiscount()
{
    DateTime start = DateTime.Now - length;

    if ((DateJoined < start) && (PreviousOrders.Count() > orderCount))
    {
        return discountPercent;
    }
    return 0;
}
```

There are many new language capabilities shown in that small code fragment. Interfaces can now include static members, including fields and methods. Different access modifiers are also enabled. The additional fields are private, the new method is public. Any of the modifiers are allowed on interface members.

Applications that use the general formula for computing the loyalty discount, but different parameters, don't need to provide a custom implementation; they can set the arguments through a static method. For example, the following code sets a "customer appreciation" that rewards any customer with more than one month's membership:

```
ICustomer.SetLoyaltyThresholds(new TimeSpan(30, 0, 0, 0), 1, 0.25m);
Console.WriteLine($"Current discount: {theCustomer.ComputeLoyaltyDiscount()}");
```

Extend the default implementation

The code you've added so far has provided a convenient implementation for those scenarios where users want something like the default implementation, or to provide an unrelated set of rules. For a final feature, let's refactor the code a bit to enable scenarios where users may want to build on the default implementation.

Consider a startup that wants to attract new customers. They offer a 50% discount off a new customer's first order. Otherwise, existing customers get the standard discount. The library author needs to move the default implementation into a `protected static` method so that any class implementing this interface can reuse the code in their implementation. The default implementation of the interface member calls this shared method as well:

```
public decimal ComputeLoyaltyDiscount() => DefaultLoyaltyDiscount(this);
protected static decimal DefaultLoyaltyDiscount(ICustomer c)
{
    DateTime start = DateTime.Now - length;

    if ((c.DateJoined < start) && (c.PreviousOrders.Count() > orderCount))
    {
        return discountPercent;
    }
    return 0;
}
```

In an implementation of a class that implements this interface, the override can call the static helper method, and extend that logic to provide the "new customer" discount:

```
public decimal ComputeLoyaltyDiscount()
{
    if (PreviousOrders.Any() == false)
        return 0.50m;
    else
        return ICustomer.DefaultLoyaltyDiscount(this);
}
```

You can see the entire finished code in our [samples repo on GitHub](#). You can get the starter application on our [samples repo on GitHub](#).

These new features mean that interfaces can be updated safely when there's a reasonable default implementation for those new members. Carefully design interfaces to express single functional ideas that can be implemented by multiple classes. That makes it easier to upgrade those interface definitions when new requirements are discovered for that same functional idea.

Tutorial: Mix functionality in when creating classes using interfaces with default interface methods

12/28/2021 • 8 minutes to read • [Edit Online](#)

Beginning with C# 8.0 on .NET Core 3.0, you can define an implementation when you declare a member of an interface. This feature provides new capabilities where you can define default implementations for features declared in interfaces. Classes can pick when to override functionality, when to use the default functionality, and when not to declare support for discrete features.

In this tutorial, you'll learn how to:

- Create interfaces with implementations that describe discrete features.
- Create classes that use the default implementations.
- Create classes that override some or all of the default implementations.

Prerequisites

You'll need to set up your machine to run .NET Core, including the C# 8.0 compiler. The C# 8.0 compiler is available starting with [Visual Studio 2019 version 16.3](#), or the [.NET Core 3.0 SDK](#) or later.

Limitations of extension methods

One way you can implement behavior that appears as part of an interface is to define [extension methods](#) that provide the default behavior. Interfaces declare a minimum set of members while providing a greater surface area for any class that implements that interface. For example, the extension methods in [Enumerable](#) provide the implementation for any sequence to be the source of a LINQ query.

Extension methods are resolved at compile time, using the declared type of the variable. Classes that implement the interface can provide a better implementation for any extension method. Variable declarations must match the implementing type to enable the compiler to choose that implementation. When the compile-time type matches the interface, method calls resolve to the extension method. Another concern with extension methods is that those methods are accessible wherever the class containing the extension methods is accessible. Classes cannot declare if they should or should not provide features declared in extension methods.

Starting with C# 8.0, you can declare the default implementations as interface methods. Then, every class automatically uses the default implementation. Any class that can provide a better implementation can override the interface method definition with a better algorithm. In one sense, this technique sounds similar to how you could use [extension methods](#).

In this article, you'll learn how default interface implementations enable new scenarios.

Design the application

Consider a home automation application. You probably have many different types of lights and indicators that could be used throughout the house. Every light must support APIs to turn them on and off, and to report the current state. Some lights and indicators may support other features, such as:

- Turn light on, then turn it off after a timer.
- Blink the light for a period of time.

Some of these extended capabilities could be emulated in devices that support the minimal set. That indicates

providing a default implementation. For those devices that have more capabilities built in, the device software would use the native capabilities. For other lights, they could choose to implement the interface and use the default implementation.

Default interface members is a better solution for this scenario than extension methods. Class authors can control which interfaces they choose to implement. Those interfaces they choose are available as methods. In addition, because default interface methods are virtual by default, the method dispatch always chooses the implementation in the class.

Let's create the code to demonstrate these differences.

Create interfaces

Start by creating the interface that defines the behavior for all lights:

```
public interface ILight
{
    void SwitchOn();
    void SwitchOff();
    bool IsOn();
}
```

A basic overhead light fixture might implement this interface as shown in the following code:

```
public class OverheadLight : ILight
{
    private bool isOn;
    public bool IsOn() => isOn;
    public void SwitchOff() => isOn = false;
    public void SwitchOn() => isOn = true;

    public override string ToString() => $"The light is {(isOn ? "on" : "off")}";
}
```

In this tutorial, the code doesn't drive IoT devices, but emulates those activities by writing messages to the console. You can explore the code without automating your house.

Next, let's define the interface for a light that can automatically turn off after a timeout:

```
public interface ITimerLight : ILight
{
    Task TurnOnFor(int duration);
}
```

You could add a basic implementation to the overhead light, but a better solution is to modify this interface definition to provide a `virtual` default implementation:

```
public interface ITimerLight : ILight
{
    public async Task TurnOnFor(int duration)
    {
        Console.WriteLine("Using the default interface method for the ITimerLight.TurnOnFor.");
        SwitchOn();
        await Task.Delay(duration);
        SwitchOff();
        Console.WriteLine("Completed ITimerLight.TurnOnFor sequence.");
    }
}
```

By adding that change, the `OverheadLight` class can implement the timer function by declaring support for the interface:

```
public class OverheadLight : ITimerLight { }
```

A different light type may support a more sophisticated protocol. It can provide its own implementation for `TurnOnFor`, as shown in the following code:

```
public class HalogenLight : ITimerLight
{
    private enum HalogenLightState
    {
        Off,
        On,
        TimerModeOn
    }

    private HalogenLightState state;
    public void SwitchOn() => state = HalogenLightState.On;
    public void SwitchOff() => state = HalogenLightState.Off;
    public bool IsOn() => state != HalogenLightState.Off;
    public async Task TurnOnFor(int duration)
    {
        Console.WriteLine("Halogen light starting timer function.");
        state = HalogenLightState.TimerModeOn;
        await Task.Delay(duration);
        state = HalogenLightState.Off;
        Console.WriteLine("Halogen light finished custom timer function");
    }

    public override string ToString() => $"The light is {state}";
}
```

Unlike overriding virtual class methods, the declaration of `TurnOnFor` in the `HalogenLight` class does not use the `override` keyword.

Mix and match capabilities

The advantages of default interface methods become clearer as you introduce more advanced capabilities. Using interfaces enables you to mix and match capabilities. It also enables each class author to choose between the default implementation and a custom implementation. Let's add an interface with a default implementation for a blinking light:

```
public interface IBlinkingLight : ILight
{
    public async Task Blink(int duration, int repeatCount)
    {
        Console.WriteLine("Using the default interface method for IBlinkingLight.Blink.");
        for (int count = 0; count < repeatCount; count++)
        {
            SwitchOn();
            await Task.Delay(duration);
            SwitchOff();
            await Task.Delay(duration);
        }
        Console.WriteLine("Done with the default interface method for IBlinkingLight.Blink.");
    }
}
```

The default implementation enables any light to blink. The overhead light can add both timer and blink

capabilities using the default implementation:

```
public class OverheadLight : ILight, ITimerLight, IBlinkingLight
{
    private bool isOn;
    public bool IsOn() => isOn;
    public void SwitchOff() => isOn = false;
    public void SwitchOn() => isOn = true;

    public override string ToString() => $"The light is {(isOn ? "on" : "off")}";
}
```

A new light type, the `LEDLight` supports both the timer function and the blink function directly. This light style implements both the `ITimerLight` and `IBlinkingLight` interfaces, and overrides the `Blink` method:

```
public class LEDLight : IBlinkingLight, ITimerLight, ILight
{
    private bool isOn;
    public void SwitchOn() => isOn = true;
    public void SwitchOff() => isOn = false;
    public bool IsOn() => isOn;
    public async Task Blink(int duration, int repeatCount)
    {
        Console.WriteLine("LED Light starting the Blink function.");
        await Task.Delay(duration * repeatCount);
        Console.WriteLine("LED Light has finished the Blink funtion.");
    }

    public override string ToString() => $"The light is {(isOn ? "on" : "off")}";
}
```

An `ExtraFancyLight` might support both blink and timer functions directly:

```
public class ExtraFancyLight : IBlinkingLight, ITimerLight, ILight
{
    private bool isOn;
    public void SwitchOn() => isOn = true;
    public void SwitchOff() => isOn = false;
    public bool IsOn() => isOn;
    public async Task Blink(int duration, int repeatCount)
    {
        Console.WriteLine("Extra Fancy Light starting the Blink function.");
        await Task.Delay(duration * repeatCount);
        Console.WriteLine("Extra Fancy Light has finished the Blink function.");
    }
    public async Task TurnOnFor(int duration)
    {
        Console.WriteLine("Extra Fancy light starting timer function.");
        await Task.Delay(duration);
        Console.WriteLine("Extra Fancy light finished custom timer function");
    }

    public override string ToString() => $"The light is {(isOn ? "on" : "off")}";
}
```

The `HalogenLight` you created earlier doesn't support blinking. So, don't add the `IBlinkingLight` to the list of its supported interfaces.

Detect the light types using pattern matching

Next, let's write some test code. You can make use of C#'s [pattern matching](#) feature to determine a light's

capabilities by examining which interfaces it supports. The following method exercises the supported capabilities of each light:

```
private static async Task TestLightCapabilities(ILight light)
{
    // Perform basic tests:
    light.SwitchOn();
    Console.WriteLine($"{Environment.NewLine}After switching on, the light is {(light.IsOn() ? "on" : "off")}");
    light.SwitchOff();
    Console.WriteLine($"{Environment.NewLine}After switching off, the light is {(light.IsOn() ? "on" : "off")}");

    if (light is ITimerLight timer)
    {
        Console.WriteLine($"{Environment.NewLine}Testing timer function");
        await timer.TurnOnFor(1000);
        Console.WriteLine($"{Environment.NewLine}Timer function completed");
    }
    else
    {
        Console.WriteLine($"{Environment.NewLine}Timer function not supported.");
    }

    if (light is IBlinkingLight blinker)
    {
        Console.WriteLine($"{Environment.NewLine}Testing blinking function");
        await blinker.Blink(500, 5);
        Console.WriteLine($"{Environment.NewLine}Blink function completed");
    }
    else
    {
        Console.WriteLine($"{Environment.NewLine}Blink function not supported.");
    }
}
```

The following code in your `Main` method creates each light type in sequence and tests that light:

```
static async Task Main(string[] args)
{
    Console.WriteLine("Testing the overhead light");
    var overhead = new OverheadLight();
    await TestLightCapabilities(overhead);
    Console.WriteLine();

    Console.WriteLine("Testing the halogen light");
    var halogen = new HalogenLight();
    await TestLightCapabilities(halogen);
    Console.WriteLine();

    Console.WriteLine("Testing the LED light");
    var led = new LEDLight();
    await TestLightCapabilities(led);
    Console.WriteLine();

    Console.WriteLine("Testing the fancy light");
    var fancy = new ExtraFancyLight();
    await TestLightCapabilities(fancy);
    Console.WriteLine();
}
```

How the compiler determines best implementation

This scenario shows a base interface without any implementations. Adding a method into the `ILight` interface introduces new complexities. The language rules governing default interface methods minimize the effect on the

concrete classes that implement multiple derived interfaces. Let's enhance the original interface with a new method to show how that changes its use. Every indicator light can report its power status as an enumerated value:

```
public enum PowerStatus
{
    NoPower,
    ACPower,
    FullBattery,
    MidBattery,
    LowBattery
}
```

The default implementation assumes no power:

```
public interface ILight
{
    void SwitchOn();
    void SwitchOff();
    bool IsOn();
    public PowerStatus Power() => PowerStatus.NoPower;
}
```

These changes compile cleanly, even though the `ExtraFancyLight` declares support for the `ILight` interface and both derived interfaces, `ITimerLight` and `IBlinkingLight`. There's only one "closest" implementation declared in the `ILight` interface. Any class that declared an override would become the one "closest" implementation. You saw examples in the preceding classes that overrode the members of other derived interfaces.

Avoid overriding the same method in multiple derived interfaces. Doing so creates an ambiguous method call whenever a class implements both derived interfaces. The compiler can't pick a single better method so it issues an error. For example, if both the `IBlinkingLight` and `ITimerLight` implemented an override of `PowerStatus`, the `OverheadLight` would need to provide a more specific override. Otherwise, the compiler can't pick between the implementations in the two derived interfaces. You can usually avoid this situation by keeping interface definitions small and focused on one feature. In this scenario, each capability of a light is its own interface; multiple interfaces are only inherited by classes.

This sample shows one scenario where you can define discrete features that can be mixed into classes. You declare any set of supported functionality by declaring which interfaces a class supports. The use of virtual default interface methods enables classes to use or define a different implementation for any or all the interface methods. This language capability provides new ways to model the real-world systems you're building. Default interface methods provide a clearer way to express related classes that may mix and match different features using virtual implementations of those capabilities.

Indices and ranges

12/28/2021 • 6 minutes to read • [Edit Online](#)

Ranges and indices provide a succinct syntax for accessing single elements or ranges in a sequence.

In this tutorial, you'll learn how to:

- Use the syntax for ranges in a sequence.
- Understand the design decisions for the start and end of each sequence.
- Learn scenarios for the [Index](#) and [Range](#) types.

Language support for indices and ranges

This language support relies on two new types and two new operators:

- [System.Index](#) represents an index into a sequence.
- The index from end operator `^`, which specifies that an index is relative to the end of a sequence.
- [System.Range](#) represents a sub range of a sequence.
- The range operator `..`, which specifies the start and end of a range as its operands.

Let's start with the rules for indices. Consider an array `sequence`. The `0` index is the same as `sequence[0]`. The `^0` index is the same as `sequence[sequence.Length]`. The expression `sequence[^0]` does throw an exception, just as `sequence[sequence.Length]` does. For any number `n`, the index `^n` is the same as `sequence[sequence.Length - n]`.

```
string[] words = new string[]
{
    "The",      // index from start  index from end
    "quick",    // 0                ^9
    "brown",    // 1                ^8
    "fox",      // 2                ^7
    "jumped",   // 3                ^6
    "over",     // 4                ^5
    "the",      // 5                ^4
    "lazy",     // 6                ^3
    "dog"       // 7                ^2
};              // 8                ^1
               // 9 (or words.Length) ^0
```

You can retrieve the last word with the `^1` index. Add the following code below the initialization:

```
Console.WriteLine($"The last word is {words[^1]}");
```

A range specifies the *start* and *end* of a range. Ranges are exclusive, meaning the *end* isn't included in the range. The range `[0..^0]` represents the entire range, just as `[0..sequence.Length]` represents the entire range.

The following code creates a subrange with the words "quick", "brown", and "fox". It includes `words[1]` through `words[3]`. The element `words[4]` isn't in the range. Add the following code to the same method. Copy and paste it at the bottom of the interactive window.

```
string[] quickBrownFox = words[1..4];
foreach (var word in quickBrownFox)
    Console.WriteLine($"< {word} >");
Console.WriteLine();
```

The following code returns the range with "lazy" and "dog". It includes `words[^2]` and `words[^1]`. The end index `words[0]` isn't included. Add the following code as well:

```
string[] lazyDog = words[^2..^0];
foreach (var word in lazyDog)
    Console.WriteLine($"< {word} >");
Console.WriteLine();
```

The following examples create ranges that are open ended for the start, end, or both:

```
string[] allWords = words[..]; // contains "The" through "dog".
string[] firstPhrase = words[..4]; // contains "The" through "fox"
string[] lastPhrase = words[6..]; // contains "the", "lazy" and "dog"
foreach (var word in allWords)
    Console.WriteLine($"< {word} >");
Console.WriteLine();
foreach (var word in firstPhrase)
    Console.WriteLine($"< {word} >");
Console.WriteLine();
foreach (var word in lastPhrase)
    Console.WriteLine($"< {word} >");
Console.WriteLine();
```

You can also declare ranges or indices as variables. The variable can then be used inside the `[` and `]` characters:

```
Index the = ^3;
Console.WriteLine(words[the]);
Range phrase = 1..4;
string[] text = words[phrase];
foreach (var word in text)
    Console.WriteLine($"< {word} >");
Console.WriteLine();
```

The following sample shows many of the reasons for those choices. Modify `x`, `y`, and `z` to try different combinations. When you experiment, use values where `x` is less than `y`, and `y` is less than `z` for valid combinations. Add the following code in a new method. Try different combinations:

```

int[] numbers = Enumerable.Range(0, 100).ToArray();
int x = 12;
int y = 25;
int z = 36;

Console.WriteLine($"{numbers[^x]} is the same as {numbers[numbers.Length - x]}");
Console.WriteLine($"{numbers[x..y].Length} is the same as {y - x}");

Console.WriteLine("numbers[x..y] and numbers[y..z] are consecutive and disjoint:");
Span<int> x_y = numbers[x..y];
Span<int> y_z = numbers[y..z];
Console.WriteLine($"{numbers[x..y] is {x_y[0]} through {x_y[^1]}, numbers[y..z] is {y_z[0]} through {y_z[^1]}");

Console.WriteLine("numbers[x..^x] removes x elements at each end:");
Span<int> x_x = numbers[x..^x];
Console.WriteLine($"{numbers[x..^x] starts with {x_x[0]} and ends with {x_x[^1]}");

Console.WriteLine("numbers[..x] means numbers[0..x] and numbers[x..] means numbers[x..^0]");
Span<int> start_x = numbers[..x];
Span<int> zero_x = numbers[0..x];
Console.WriteLine($"{start_x[0]}..{start_x[^1]} is the same as {zero_x[0]}..{zero_x[^1]}");
Span<int> z_end = numbers[z..];
Span<int> z_zero = numbers[z..^0];
Console.WriteLine($"{z_end[0]}..{z_end[^1]} is the same as {z_zero[0]}..{z_zero[^1]}");

```

Type support for indices and ranges

Indexes and ranges provide clear, concise syntax to access a single element or a range of elements in a sequence. An index expression typically returns the type of the elements of a sequence. A range expression typically returns the same sequence type as the source sequence.

Any type that provides an [indexer](#) with an [Index](#) or [Range](#) parameter explicitly supports indices or ranges respectively. An indexer that takes a single [Range](#) parameter may return a different sequence type, such as [System.Span<T>](#).

IMPORTANT

The performance of code using the range operator depends on the type of the sequence operand.

The time complexity of the range operator depends on the sequence type. For example, if the sequence is a `string` or an array, then the result is a copy of the specified section of the input, so the time complexity is $O(N)$ (where N is the length of the range). On the other hand, if it's a [System.Span<T>](#) or a [System.Memory<T>](#), the result references the same backing store, which means there is no copy and the operation is $O(1)$.

In addition to the time complexity, this causes extra allocations and copies, impacting performance. In performance sensitive code, consider using `Span<T>` or `Memory<T>` as the sequence type, since the range operator does not allocate for them.

A type is **countable** if it has a property named `Length` or `Count` with an accessible getter and a return type of `int`. A countable type that doesn't explicitly support indices or ranges may provide an implicit support for them. For more information, see the [Implicit Index support](#) and [Implicit Range support](#) sections of the [feature proposal note](#). Ranges using implicit range support return the same sequence type as the source sequence.

For example, the following .NET types support both indices and ranges: [String](#), [Span<T>](#), and [ReadOnlySpan<T>](#). The [List<T>](#) supports indices but doesn't support ranges.

[Array](#) has more nuanced behavior. Single dimension arrays support both indices and ranges. Multi-dimensional arrays don't support indexers or ranges. The indexer for a multi-dimensional array has multiple parameters, not

a single parameter. Jagged arrays, also referred to as an array of arrays, support both ranges and indexers. The following example shows how to iterate a rectangular subsection of a jagged array. It iterates the section in the center, excluding the first and last three rows, and the first and last two columns from each selected row:

```
var jagged = new int[10][]
{
    new int[10] { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9},
    new int[10] { 10,11,12,13,14,15,16,17,18,19},
    new int[10] { 20,21,22,23,24,25,26,27,28,29},
    new int[10] { 30,31,32,33,34,35,36,37,38,39},
    new int[10] { 40,41,42,43,44,45,46,47,48,49},
    new int[10] { 50,51,52,53,54,55,56,57,58,59},
    new int[10] { 60,61,62,63,64,65,66,67,68,69},
    new int[10] { 70,71,72,73,74,75,76,77,78,79},
    new int[10] { 80,81,82,83,84,85,86,87,88,89},
    new int[10] { 90,91,92,93,94,95,96,97,98,99},
};

var selectedRows = jagged[3..^3];

foreach (var row in selectedRows)
{
    var selectedColumns = row[2..^2];
    foreach (var cell in selectedColumns)
    {
        Console.Write($"{cell}, ");
    }
    Console.WriteLine();
}
```

In all cases, the range operator for [Array](#) allocates an array to store the elements returned.

Scenarios for indices and ranges

You'll often use ranges and indices when you want to analyze a portion of a larger sequence. The new syntax is clearer in reading exactly what portion of the sequence is involved. The local function `MovingAverage` takes a [Range](#) as its argument. The method then enumerates just that range when calculating the min, max, and average. Try the following code in your project:

```

int[] sequence = Sequence(1000);

for(int start = 0; start < sequence.Length; start += 100)
{
    Range r = start..(start+10);
    var (min, max, average) = MovingAverage(sequence, r);
    Console.WriteLine($"From {r.Start} to {r.End}:    \tMin: {min},\tMax: {max},\tAverage: {average}");
}

for (int start = 0; start < sequence.Length; start += 100)
{
    Range r = ^(start + 10)..^start;
    var (min, max, average) = MovingAverage(sequence, r);
    Console.WriteLine($"From {r.Start} to {r.End}:  \tMin: {min},\tMax: {max},\tAverage: {average}");
}

(int min, int max, double average) MovingAverage(int[] subSequence, Range range) =>
(
    subSequence[range].Min(),
    subSequence[range].Max(),
    subSequence[range].Average()
);

int[] Sequence(int count) =>
    Enumerable.Range(0, count).Select(x => (int)(Math.Sqrt(x) * 100)).ToArray();

```

Tutorial: Express your design intent more clearly with nullable and non-nullable reference types

12/28/2021 • 10 minutes to read • [Edit Online](#)

C# 8.0 introduces [nullable reference types](#), which complement reference types the same way nullable value types complement value types. You declare a variable to be a **nullable reference type** by appending a `?` to the type. For example, `string?` represents a nullable `string`. You can use these new types to more clearly express your design intent: some variables *must always have a value*, others *may be missing a value*.

In this tutorial, you'll learn how to:

- Incorporate nullable and non-nullable reference types into your designs
- Enable nullable reference type checks throughout your code.
- Write code where the compiler enforces those design decisions.
- Use the nullable reference feature in your own designs

Prerequisites

You'll need to set up your machine to run .NET Core, including the C# 8.0 compiler. The C# 8.0 compiler is available with [Visual Studio 2019](#), or [.NET Core 3.0](#).

This tutorial assumes you're familiar with C# and .NET, including either Visual Studio or the .NET CLI.

Incorporate nullable reference types into your designs

In this tutorial, you'll build a library that models running a survey. The code uses both nullable reference types and non-nullable reference types to represent the real-world concepts. The survey questions can never be null. A respondent might prefer not to answer a question. The responses might be `null` in this case.

The code you'll write for this sample expresses that intent, and the compiler enforces that intent.

Create the application and enable nullable reference types

Create a new console application either in Visual Studio or from the command line using `dotnet new console`. Name the application `NullableIntroduction`. Once you've created the application, you'll need to specify that the entire project compiles in an enabled **nullable annotation context**. Open the `.csproj` file and add a `Nullable` element to the `PropertyGroup` element. Set its value to `enable`. You must opt into the **nullable reference types** feature, even in C# 8.0 projects. That's because once the feature is turned on, existing reference variable declarations become **non-nullable reference types**. While that decision will help find issues where existing code may not have proper null-checks, it may not accurately reflect your original design intent:

```
<Nullable>enable</Nullable>
```

Prior to .NET 6, new projects do not include the `Nullable` element. Beginning with .NET 6, new projects include the `<Nullable>enable</Nullable>` element in the project file.

Design the types for the application

This survey application requires creating a number of classes:

- A class that models the list of questions.

- A class that models a list of people contacted for the survey.
- A class that models the answers from a person that took the survey.

These types will make use of both nullable and non-nullable reference types to express which members are required and which members are optional. Nullable reference types communicate that design intent clearly:

- The questions that are part of the survey can never be null: It makes no sense to ask an empty question.
- The respondents can never be null. You'll want to track people you contacted, even respondents that declined to participate.
- Any response to a question may be null. Respondents can decline to answer some or all questions.

If you've programmed in C#, you may be so accustomed to reference types that allow `null` values that you may have missed other opportunities to declare non-nullable instances:

- The collection of questions should be non-nullable.
- The collection of respondents should be non-nullable.

As you write the code, you'll see that a non-nullable reference type as the default for references avoids common mistakes that could lead to [NullReferenceExceptions](#). One lesson from this tutorial is that you made decisions about which variables could or could not be `null`. The language didn't provide syntax to express those decisions. Now it does.

The app you'll build does the following steps:

1. Creates a survey and adds questions to it.
2. Creates a pseudo-random set of respondents for the survey.
3. Contacts respondents until the completed survey size reaches the goal number.
4. Writes out important statistics on the survey responses.

Build the survey with nullable and non-nullable reference types

The first code you'll write creates the survey. You'll write classes to model a survey question and a survey run. Your survey has three types of questions, distinguished by the format of the answer: Yes/No answers, number answers, and text answers. Create a `public SurveyQuestion` class:

```
namespace NullableIntroduction
{
    public class SurveyQuestion
    {
    }
}
```

The compiler interprets every reference type variable declaration as a **non-nullable** reference type for code in an enabled nullable annotation context. You can see your first warning by adding properties for the question text and the type of question, as shown in the following code:

```

namespace NullableIntroduction
{
    public enum QuestionType
    {
        YesNo,
        Number,
        Text
    }

    public class SurveyQuestion
    {
        public string QuestionText { get; }
        public QuestionType TypeOfQuestion { get; }
    }
}

```

Because you haven't initialized `QuestionText`, the compiler issues a warning that a non-nullable property hasn't been initialized. Your design requires the question text to be non-null, so you add a constructor to initialize it and the `QuestionType` value as well. The finished class definition looks like the following code:

```

namespace NullableIntroduction
{
    public enum QuestionType
    {
        YesNo,
        Number,
        Text
    }

    public class SurveyQuestion
    {
        public string QuestionText { get; }
        public QuestionType TypeOfQuestion { get; }

        public SurveyQuestion(QuestionType typeOfQuestion, string text) =>
            (TypeOfQuestion, QuestionText) = (typeOfQuestion, text);
    }
}

```

Adding the constructor removes the warning. The constructor argument is also a non-nullable reference type, so the compiler doesn't issue any warnings.

Next, create a `public` class named `SurveyRun`. This class contains a list of `SurveyQuestion` objects and methods to add questions to the survey, as shown in the following code:

```

using System.Collections.Generic;

namespace NullableIntroduction
{
    public class SurveyRun
    {
        private List<SurveyQuestion> surveyQuestions = new List<SurveyQuestion>();

        public void AddQuestion(QuestionType type, string question) =>
            AddQuestion(new SurveyQuestion(type, question));
        public void AddQuestion(SurveyQuestion surveyQuestion) => surveyQuestions.Add(surveyQuestion);
    }
}

```

As before, you must initialize the list object to a non-null value or the compiler issues a warning. There are no null checks in the second overload of `AddQuestion` because they aren't needed: You've declared that variable to

be non-nullable. Its value can't be `null`.

Switch to *Program.cs* in your editor and replace the contents of `Main` with the following lines of code:

```
var surveyRun = new SurveyRun();
surveyRun.AddQuestion(QuestionType.YesNo, "Has your code ever thrown a NullReferenceException?");
surveyRun.AddQuestion(new SurveyQuestion(QuestionType.Number, "How many times (to the nearest 100) has that happened?"));
surveyRun.AddQuestion(QuestionType.Text, "What is your favorite color?");
```

Because the entire project is in an enabled nullable annotation context, you'll get warnings when you pass `null` to any method expecting a non-nullable reference type. Try it by adding the following line to `Main`:

```
surveyRun.AddQuestion(QuestionType.Text, default);
```

Create respondents and get answers to the survey

Next, write the code that generates answers to the survey. This process involves several small tasks:

1. Build a method that generates respondent objects. These represent people asked to fill out the survey.
2. Build logic to simulate asking the questions to a respondent and collecting answers or noting that a respondent didn't answer.
3. Repeat until enough respondents have answered the survey.

You'll need a class to represent a survey response, so add that now. Enable nullable support. Add an `Id` property and a constructor that initializes it, as shown in the following code:

```
namespace NullableIntroduction
{
    public class SurveyResponse
    {
        public int Id { get; }

        public SurveyResponse(int id) => Id = id;
    }
}
```

Next, add a `static` method to create new participants by generating a random ID:

```
private static readonly Random randomGenerator = new Random();
public static SurveyResponse GetRandomId() => new SurveyResponse(randomGenerator.Next());
```

The main responsibility of this class is to generate the responses for a participant to the questions in the survey. This responsibility has a few steps:

1. Ask for participation in the survey. If the person doesn't consent, return a missing (or null) response.
2. Ask each question and record the answer. Each answer may also be missing (or null).

Add the following code to your `SurveyResponse` class:

```

private Dictionary<int, string>? surveyResponses;
public bool AnswerSurvey(IEnumerable<SurveyQuestion> questions)
{
    if (ConsentToSurvey())
    {
        surveyResponses = new Dictionary<int, string>();
        int index = 0;
        foreach (var question in questions)
        {
            var answer = GenerateAnswer(question);
            if (answer != null)
            {
                surveyResponses.Add(index, answer);
            }
            index++;
        }
    }
    return surveyResponses != null;
}

private bool ConsentToSurvey() => randomGenerator.Next(0, 2) == 1;

private string? GenerateAnswer(SurveyQuestion question)
{
    switch (question.TypeOfQuestion)
    {
        case QuestionType.YesNo:
            int n = randomGenerator.Next(-1, 2);
            return (n == -1) ? default : (n == 0) ? "No" : "Yes";
        case QuestionType.Number:
            n = randomGenerator.Next(-30, 101);
            return (n < 0) ? default : n.ToString();
        case QuestionType.Text:
        default:
            switch (randomGenerator.Next(0, 5))
            {
                case 0:
                    return default;
                case 1:
                    return "Red";
                case 2:
                    return "Green";
                case 3:
                    return "Blue";
            }
            return "Red. No, Green. Wait.. Blue... AAARGGGGGHHH!";
    }
}

```

The storage for the survey answers is a `Dictionary<int, string>?`, indicating that it may be null. You're using the new language feature to declare your design intent, both to the compiler and to anyone reading your code later. If you ever dereference `surveyResponses` without checking for the `null` value first, you'll get a compiler warning. You don't get a warning in the `AnswerSurvey` method because the compiler can determine the `surveyResponses` variable was set to a non-null value above.

Using `null` for missing answers highlights a key point for working with nullable reference types: your goal isn't to remove all `null` values from your program. Rather, your goal is to ensure that the code you write expresses the intent of your design. Missing values are a necessary concept to express in your code. The `null` value is a clear way to express those missing values. Trying to remove all `null` values only leads to defining some other way to express those missing values without `null`.

Next, you need to write the `PerformSurvey` method in the `SurveyRun` class. Add the following code in the `SurveyRun` class:

```
private List<SurveyResponse>? respondents;
public void PerformSurvey(int numberOfRespondents)
{
    int respondentsConsenting = 0;
    respondents = new List<SurveyResponse>();
    while (respondentsConsenting < numberOfRespondents)
    {
        var respondent = SurveyResponse.GetRandomId();
        if (respondent.AnswerSurvey(surveyQuestions))
            respondentsConsenting++;
        respondents.Add(respondent);
    }
}
```

Here again, your choice of a nullable `List<SurveyResponse>?` indicates the response may be null. That indicates the survey hasn't been given to any respondents yet. Notice that respondents are added until enough have consented.

The last step to run the survey is to add a call to perform the survey at the end of the `Main` method:

```
surveyRun.PerformSurvey(50);
```

Examine survey responses

The last step is to display survey results. You'll add code to many of the classes you've written. This code demonstrates the value of distinguishing nullable and non-nullable reference types. Start by adding the following two expression-bodied members to the `SurveyResponse` class:

```
public bool AnsweredSurvey => surveyResponses != null;
public string Answer(int index) => surveyResponses?.GetValueOrDefault(index) ?? "No answer";
```

Because `surveyResponses` is a nullable reference type, null checks are necessary before de-referencing it. The `Answer` method returns a non-nullable string, so we have to cover the case of a missing answer by using the null-coalescing operator.

Next, add these three expression-bodied members to the `SurveyRun` class:

```
public IEnumerable<SurveyResponse> AllParticipants => (respondents ?? Enumerable.Empty<SurveyResponse>());
public ICollection<SurveyQuestion> Questions => surveyQuestions;
public SurveyQuestion GetQuestion(int index) => surveyQuestions[index];
```

The `AllParticipants` member must take into account that the `respondents` variable might be null, but the return value can't be null. If you change that expression by removing the `??` and the empty sequence that follows, the compiler warns you the method might return `null` and its return signature returns a non-nullable type.

Finally, add the following loop at the bottom of the `Main` method:

```

foreach (var participant in surveyRun.AllParticipants)
{
    Console.WriteLine($"Participant: {participant.Id}:");
    if (participant.AnsweredSurvey)
    {
        for (int i = 0; i < surveyRun.Questions.Count; i++)
        {
            var answer = participant.Answer(i);
            Console.WriteLine($"  {surveyRun.GetQuestion(i).QuestionText} : {answer}");
        }
    }
    else
    {
        Console.WriteLine($"  No responses");
    }
}

```

You don't need any `null` checks in this code because you've designed the underlying interfaces so that they all return non-nullable reference types.

Get the code

You can get the code for the finished tutorial from our [samples](#) repository in the [csharp/NullableIntroduction](#) folder.

Experiment by changing the type declarations between nullable and non-nullable reference types. See how that generates different warnings to ensure you don't accidentally dereference a `null`.

Next steps

Learn how to use nullable reference type when using Entity Framework:

[Entity Framework Core Fundamentals: Working with Nullable Reference Types](#)

Tutorial: Generate and consume async streams using C# 8.0 and .NET Core 3.0

12/28/2021 • 9 minutes to read • [Edit Online](#)

C# 8.0 introduces **async streams**, which model a streaming source of data. Data streams often retrieve or generate elements asynchronously. Async streams rely on new interfaces introduced in .NET Standard 2.1. These interfaces are supported in .NET Core 3.0 and later. They provide a natural programming model for asynchronous streaming data sources.

In this tutorial, you'll learn how to:

- Create a data source that generates a sequence of data elements asynchronously.
- Consume that data source asynchronously.
- Support cancellation and captured contexts for asynchronous streams.
- Recognize when the new interface and data source are preferred to earlier synchronous data sequences.

Prerequisites

You'll need to set up your machine to run .NET Core, including the C# 8.0 compiler. The C# 8 compiler is available starting with [Visual Studio 2019 version 16.3](#) or [.NET Core 3.0 SDK](#).

You'll need to create a [GitHub access token](#) so that you can access the GitHub GraphQL endpoint. Select the following permissions for your GitHub Access Token:

- repo:status
- public_repo

Save the access token in a safe place so you can use it to gain access to the GitHub API endpoint.

WARNING

Keep your personal access token secure. Any software with your personal access token could make GitHub API calls using your access rights.

This tutorial assumes you're familiar with C# and .NET, including either Visual Studio or the .NET CLI.

Run the starter application

You can get the code for the starter application used in this tutorial from the [dotnet/docs](#) repository in the [csharp/whats-new/tutorials](#) folder.

The starter application is a console application that uses the [GitHub GraphQL](#) interface to retrieve recent issues written in the [dotnet/docs](#) repository. Start by looking at the following code for the starter app `Main` method:

```

static async Task Main(string[] args)
{
    //Follow these steps to create a GitHub Access Token
    // https://help.github.com/articles/creating-a-personal-access-token-for-the-command-line/#creating-a-token
    //Select the following permissions for your GitHub Access Token:
    // - repo:status
    // - public_repo
    // Replace the 3rd parameter to the following code with your GitHub access token.
    var key = GetEnvVariable("GitHubKey",
        "You must store your GitHub key in the 'GitHubKey' environment variable",
        "");

    var client = new GitHubClient(new Octokit.ProductHeaderValue("IssueQueryDemo"))
    {
        Credentials = new Octokit.Credentials(key)
    };

    var progressReporter = new progressStatus((num) =>
    {
        Console.WriteLine($"Received {num} issues in total");
    });
    CancellationTokensource cancellationSource = new CancellationTokensource();

    try
    {
        var results = await RunPagedQueryAsync(client, PagedIssueQuery, "docs",
            cancellationSource.Token, progressReporter);
        foreach(var issue in results)
            Console.WriteLine(issue);
    }
    catch (OperationCanceledException)
    {
        Console.WriteLine("Work has been cancelled");
    }
}

```

You can either set a `GitHubKey` environment variable to your personal access token, or you can replace the last argument in the call to `GetEnvVariable` with your personal access token. Don't put your access code in source code if you'll be sharing the source with others. Never upload access codes to a shared source repository.

After creating the GitHub client, the code in `Main` creates a progress reporting object and a cancellation token. Once those objects are created, `Main` calls `RunPagedQueryAsync` to retrieve the most recent 250 created issues. After that task has finished, the results are displayed.

When you run the starter application, you can make some important observations about how this application runs. You'll see progress reported for each page returned from GitHub. You can observe a noticeable pause before GitHub returns each new page of issues. Finally, the issues are displayed only after all 10 pages have been retrieved from GitHub.

Examine the implementation

The implementation reveals why you observed the behavior discussed in the previous section. Examine the code for `RunPagedQueryAsync`:


```

private static async Task<JArray> RunPagedQueryAsync(GitHubClient client, string queryText, string repoName,
Cancellation token cancel, IProgress<int> progress)
{
    var issueAndPRQuery = new GraphQLRequest
    {
        Query = queryText
    };
    issueAndPRQuery.Variables["repo_name"] = repoName;

    JArray finalResults = new JArray();
    bool hasMorePages = true;
    int pagesReturned = 0;
    int issuesReturned = 0;

    // Stop with 10 pages, because these are large repos:
    while (hasMorePages && (pagesReturned++ < 10))
    {
        var postBody = issueAndPRQuery.ToJsonText();
        var response = await client.Connection.Post<string>(new Uri("https://api.github.com/graphql"),
            postBody, "application/json", "application/json");

        JObject results = JObject.Parse(response.HttpResponse.Body.ToString());

        int totalCount = (int)issues(results)["totalCount"];
        hasMorePages = (bool)pageInfo(results)["hasPreviousPage"];
        issueAndPRQuery.Variables["start_cursor"] = pageInfo(results)["startCursor"].ToString();
        issuesReturned += issues(results)["nodes"].Count();
        finalResults.Merge(issues(results)["nodes"]);
        progress?.Report(issuesReturned);
        cancel.ThrowIfCancellationRequested();
    }
    return finalResults;

    JObject issues(JObject result) => (JObject)result["data"]["repository"]["issues"];
    JObject pageInfo(JObject result) => (JObject)issues(result)["pageInfo"];
}

```

Let's concentrate on the paging algorithm and async structure of the preceding code. (You can consult the [GitHub GraphQL documentation](#) for details on the GitHub GraphQL API.) The `RunPagedQueryAsync` method enumerates the issues from most recent to oldest. It requests 25 issues per page and examines the `pageInfo` structure of the response to continue with the previous page. That follows GraphQL's standard paging support for multi-page responses. The response includes a `pageInfo` object that includes a `hasPreviousPages` value and a `startCursor` value used to request the previous page. The issues are in the `nodes` array. The `RunPagedQueryAsync` method appends these nodes to an array that contains all the results from all pages.

After retrieving and restoring a page of results, `RunPagedQueryAsync` reports progress and checks for cancellation. If cancellation has been requested, `RunPagedQueryAsync` throws an [OperationCanceledException](#).

There are several elements in this code that can be improved. Most importantly, `RunPagedQueryAsync` must allocate storage for all the issues returned. This sample stops at 250 issues because retrieving all open issues would require much more memory to store all the retrieved issues. The protocols for supporting progress reports and cancellation make the algorithm harder to understand on its first reading. More types and APIs are involved. You must trace the communications through the [CancellationTokenSource](#) and its associated [CancellationToken](#) to understand where cancellation is requested and where it's granted.

Async streams provide a better way

Async streams and the associated language support address all those concerns. The code that generates the sequence can now use `yield return` to return elements in a method that was declared with the `async` modifier. You can consume an async stream using an `await foreach` loop just as you consume any sequence using a

`foreach` loop.

These new language features depend on three new interfaces added to .NET Standard 2.1 and implemented in .NET Core 3.0:

- [System.Collections.Generic.IAsyncEnumerable<T>](#)
- [System.Collections.Generic.IAsyncEnumerator<T>](#)
- [System.IAsyncDisposable](#)

These three interfaces should be familiar to most C# developers. They behave in a manner similar to their synchronous counterparts:

- [System.Collections.Generic.IEnumerable<T>](#)
- [System.Collections.Generic.IEnumerator<T>](#)
- [System.IDisposable](#)

One type that may be unfamiliar is [System.Threading.Tasks.ValueTask](#). The `ValueTask` struct provides a similar API to the [System.Threading.Tasks.Task](#) class. `ValueTask` is used in these interfaces for performance reasons.

Convert to async streams

Next, convert the `RunPagedQueryAsync` method to generate an async stream. First, change the signature of `RunPagedQueryAsync` to return an `IAsyncEnumerable<JToken>`, and remove the cancellation token and progress objects from the parameter list as shown in the following code:

```
private static async IAsyncEnumerable<JToken> RunPagedQueryAsync(GitHubClient client,
    string queryText, string repoName)
```

The starter code processes each page as the page is retrieved, as shown in the following code:

```
finalResults.Merge(issues(results)["nodes"]);
progress?.Report(issuesReturned);
cancel.ThrowIfCancellationRequested();
```

Replace those three lines with the following code:

```
foreach (JObject issue in issues(results)["nodes"])
    yield return issue;
```

You can also remove the declaration of `finalResults` earlier in this method and the `return` statement that follows the loop you modified.

You've finished the changes to generate an async stream. The finished method should resemble the following code:

```

private static async IEnumerable<JToken> RunPagedQueryAsync(GitHubClient client,
    string queryText, string repoName)
{
    var issueAndPRQuery = new GraphQLRequest
    {
        Query = queryText
    };
    issueAndPRQuery.Variables["repo_name"] = repoName;

    bool hasMorePages = true;
    int pagesReturned = 0;
    int issuesReturned = 0;

    // Stop with 10 pages, because these are large repos:
    while (hasMorePages && (pagesReturned++ < 10))
    {
        var postBody = issueAndPRQuery.ToJsonText();
        var response = await client.Connection.Post<string>(new Uri("https://api.github.com/graphql"),
            postBody, "application/json", "application/json");

        JObject results = JObject.Parse(response.HttpResponse.Body.ToString());

        int totalCount = (int)issues(results)["totalCount"];
        hasMorePages = (bool)pageInfo(results)["hasPreviousPage"];
        issueAndPRQuery.Variables["start_cursor"] = pageInfo(results)["startCursor"].ToString();
        issuesReturned += issues(results)["nodes"].Count();

        foreach (JObject issue in issues(results)["nodes"])
            yield return issue;
    }

    JObject issues(JObject result) => (JObject)result["data"]["repository"]["issues"];
    JObject pageInfo(JObject result) => (JObject)issues(result)["pageInfo"];
}

```

Next, you change the code that consumes the collection to consume the async stream. Find the following code in `Main` that processes the collection of issues:

```

var progressReporter = new progressStatus((num) =>
{
    Console.WriteLine($"Received {num} issues in total");
});
CancellationTokenSource cancellationSource = new CancellationTokenSource();

try
{
    var results = await RunPagedQueryAsync(client, PagedIssueQuery, "docs",
        cancellationSource.Token, progressReporter);
    foreach(var issue in results)
        Console.WriteLine(issue);
}
catch (OperationCanceledException)
{
    Console.WriteLine("Work has been cancelled");
}

```

Replace that code with the following `await foreach` loop:

```
int num = 0;
await foreach (var issue in RunPagedQueryAsync(client, PagedIssueQuery, "docs"))
{
    Console.WriteLine(issue);
    Console.WriteLine($"Received {++num} issues in total");
}
```

The new interface [IAsyncEnumerator<T>](#) derives from [IAsyncDisposable](#). That means the preceding loop will asynchronously dispose the stream when the loop finishes. You can imagine the loop looks like the following code:

```
int num = 0;
var enumerator = RunPagedQueryAsync(client, PagedIssueQuery, "docs").GetEnumeratorAsync();
try
{
    while (await enumerator.MoveNextAsync())
    {
        var issue = enumerator.Current;
        Console.WriteLine(issue);
        Console.WriteLine($"Received {++num} issues in total");
    }
} finally
{
    if (enumerator != null)
        await enumerator.DisposeAsync();
}
```

By default, stream elements are processed in the captured context. If you want to disable capturing of the context, use the [TaskAsyncEnumerableExtensions.ConfigureAwait](#) extension method. For more information about synchronization contexts and capturing the current context, see the article on [consuming the Task-based asynchronous pattern](#).

Async streams support cancellation using the same protocol as other `async` methods. You would modify the signature for the async iterator method as follows to support cancellation:

```

private static async IEnumerable<JToken> RunPagedQueryAsync(GitHubClient client,
    string queryText, string repoName, [EnumeratorCancellation] CancellationToken cancellationToken =
default)
{
    var issueAndPRQuery = new GraphQLRequest
    {
        Query = queryText
    };
    issueAndPRQuery.Variables["repo_name"] = repoName;

    bool hasMorePages = true;
    int pagesReturned = 0;
    int issuesReturned = 0;

    // Stop with 10 pages, because these are large repos:
    while (hasMorePages && (pagesReturned++ < 10))
    {
        var postBody = issueAndPRQuery.ToJsonText();
        var response = await client.Connection.Post<string>(new Uri("https://api.github.com/graphql"),
            postBody, "application/json", "application/json");

        JObject results = JObject.Parse(response.HttpResponse.Body.ToString());

        int totalCount = (int)issues(results)["totalCount"];
        hasMorePages = (bool)pageInfo(results)["hasPreviousPage"];
        issueAndPRQuery.Variables["start_cursor"] = pageInfo(results)["startCursor"].ToString();
        issuesReturned += issues(results)["nodes"].Count();

        foreach (JObject issue in issues(results)["nodes"])
            yield return issue;
    }

    JObject issues(JObject result) => (JObject)result["data"]["repository"]["issues"];
    JObject pageInfo(JObject result) => (JObject)issues(result)["pageInfo"];
}

```

The [EnumeratorCancellationAttribute](#) attribute causes the compiler to generate code for the [IAsyncEnumerator<T>](#) that makes the token passed to `GetAsyncEnumerator` visible to the body of the async iterator as that argument. Inside `runQueryAsync`, you could examine the state of the token and cancel further work if requested.

You use another extension method, [WithCancellation](#), to pass the cancellation token to the async stream. You would modify the loop enumerating the issues as follows:

```

private static async Task EnumerateWithCancellation(GitHubClient client)
{
    int num = 0;
    var cancellation = new CancellationTokenSource();
    await foreach (var issue in RunPagedQueryAsync(client, PagedIssueQuery, "docs")
        .WithCancellation(cancellation.Token))
    {
        Console.WriteLine(issue);
        Console.WriteLine($"Received {++num} issues in total");
    }
}

```

You can get the code for the finished tutorial from the [dotnet/docs](#) repository in the [csharp/whats-new/tutorials](#) folder.

Run the finished application

Run the application again. Contrast its behavior with the behavior of the starter application. The first page of

results is enumerated as soon as it's available. There's an observable pause as each new page is requested and retrieved, then the next page's results are quickly enumerated. The `try` / `catch` block isn't needed to handle cancellation: the caller can stop enumerating the collection. Progress is clearly reported because the async stream generates results as each page is downloaded. The status for each issue returned is seamlessly included in the `await foreach` loop. You don't need a callback object to track progress.

You can see improvements in memory use by examining the code. You no longer need to allocate a collection to store all the results before they're enumerated. The caller can determine how to consume the results and if a storage collection is needed.

Run both the starter and finished applications and you can observe the differences between the implementations for yourself. You can delete the GitHub access token you created when you started this tutorial after you've finished. If an attacker gained access to that token, they could access GitHub APIs using your credentials.

Tutorial: Write a custom string interpolation handler

12/28/2021 • 12 minutes to read • [Edit Online](#)

In this tutorial, you'll learn how to:

- Implement the string interpolation handler pattern
- Interact with the receiver in a string interpolation operation.
- Add arguments to the string interpolation handler
- Understand the new library features for string interpolation

Prerequisites

You'll need to set up your machine to run .NET 6, including the C# 10 compiler. The C# 10 compiler is available starting with [Visual Studio 2022](#) or [.NET 6 SDK](#).

This tutorial assumes you're familiar with C# and .NET, including either Visual Studio or the .NET CLI.

New outline

C# 10 adds support for a custom *interpolated string handler*. An interpolated string handler is a type that processes the placeholder expression in an interpolated string. Without a custom handler, placeholders are processed similar to [String.Format](#). Each placeholder is formatted as text, and then the components are concatenated to form the resulting string.

You can write a handler for any scenario where you use information about the resulting string. Will it be used? What constraints are on the format? Some examples include:

- You may require none of the resulting strings are greater than some limit, such as 80 characters. You can process the interpolated strings to fill a fixed-length buffer, and stop processing once that buffer length is reached.
- You may have a tabular format, and each placeholder must have a fixed length. A custom handler can enforce that, rather than forcing all client code to conform.

In this tutorial, you'll create a string interpolation handler for one of the core performance scenarios: logging libraries. Depending on the configured log level, the work to construct a log message isn't needed. If logging is off, the work to construct a string from an interpolated string expression isn't needed. The message is never printed, so any string concatenation can be skipped. In addition, any expressions used in the placeholders, including generating stack traces, doesn't need to be done.

An interpolated string handler can determine if the formatted string will be used, and only perform the necessary work if needed.

Initial implementation

Let's start from a basic `Logger` class that supports different levels:

```

public enum LogLevel
{
    Off,
    Critical,
    Error,
    Warning,
    Information,
    Trace
}

public class Logger
{
    public LogLevel EnabledLevel { get; init; } = LogLevel.Error;

    public void LogMessage(LogLevel level, string msg)
    {
        if (EnabledLevel < level) return;
        Console.WriteLine(msg);
    }
}

```

This `Logger` supports six different levels. When a message won't pass the log level filter, there's no output. The public API for the logger accepts a (fully formatted) string as the message. All the work to create the string has already been done.

Implement the handler pattern

This step is to build an *interpolated string handler* that recreates the current behavior. An interpolated string handler is a type that must have the following characteristics:

- The [System.Runtime.CompilerServices.InterpolatedStringHandlerAttribute](#) applied to the type.
- A constructor that has two `int` parameters, `literalLength` and `formatCount`. (More parameters are allowed).
- A public `AppendLiteral` method with the signature: `public void AppendLiteral(string s)`.
- A generic public `AppendFormatted` method with the signature: `public void AppendFormatted<T>(T t)`.

Internally, the builder creates the formatted string, and provides a member for a client to retrieve that string. The following code shows a `LogInterpolatedStringHandler` type that meets these requirements:


```

[InterpolatedStringHandler]
public ref struct LogInterpolatedStringHandler
{
    // Storage for the built-up string
    StringBuilder builder;

    public LogInterpolatedStringHandler(int literalLength, int formattedCount)
    {
        builder = new StringBuilder(literalLength);
        Console.WriteLine($"\\tliteral length: {literalLength}, formattedCount: {formattedCount}");
    }

    public void AppendLiteral(string s)
    {
        Console.WriteLine($"\\tAppendLiteral called: {{{s}}}");

        builder.Append(s);
        Console.WriteLine($"\\tAppended the literal string");
    }

    public void AppendFormatted<T>(T t)
    {
        Console.WriteLine($"\\tAppendFormatted called: {{{t}}} is of type {typeof(T)}");

        builder.Append(t?.ToString());
        Console.WriteLine($"\\tAppended the formatted object");
    }

    internal string GetFormattedText() => builder.ToString();
}

```

You can now add an overload to `LogMessage` in the `Logger` class to try your new interpolated string handler:

```

public void LogMessage(LogLevel level, LogInterpolatedStringHandler builder)
{
    if (EnabledLevel < level) return;
    Console.WriteLine(builder.GetFormattedText());
}

```

You don't need to remove the original `LogMessage` method, the compiler will prefer a method with an interpolated handler parameter over a method with a `string` parameter when the argument is an interpolated string expression.

You can verify that the new handler is invoked using the following code as the main program:

```

var logger = new Logger() { EnabledLevel = LogLevel.Warning };
var time = DateTime.Now;

logger.LogMessage(LogLevel.Error, $"Error Level. CurrentTime: {time}. This is an error. It will be printed.");
logger.LogMessage(LogLevel.Trace, $"Trace Level. CurrentTime: {time}. This won't be printed.");
logger.LogMessage(LogLevel.Warning, "Warning Level. This warning is a string, not an interpolated string expression.");

```

Running the application produces output similar to the following text:

```

    literal length: 65, formattedCount: 1
    AppendLiteral called: {Error Level. CurrentTime: }
    Appended the literal string
    AppendFormatted called: {10/20/2021 12:19:10 PM} is of type System.DateTime
    Appended the formatted object
    AppendLiteral called: {. This is an error. It will be printed.}
    Appended the literal string
Error Level. CurrentTime: 10/20/2021 12:19:10 PM. This is an error. It will be printed.
    literal length: 50, formattedCount: 1
    AppendLiteral called: {Trace Level. CurrentTime: }
    Appended the literal string
    AppendFormatted called: {10/20/2021 12:19:10 PM} is of type System.DateTime
    Appended the formatted object
    AppendLiteral called: {. This won't be printed.}
    Appended the literal string
Warning Level. This warning is a string, not an interpolated string expression.

```

Tracing through the output, you can see how the compiler adds code to call the handler and build the string:

- The compiler adds a call to construct the handler, passing the total length of the literal text in the format string, and the number of placeholders.
- The compiler adds calls to `AppendLiteral` and `AppendFormatted` for each section of the literal string and for each placeholder.
- The compiler invokes the `LogMessage` method using the `CoreInterpolatedStringHandler` as the argument.

Finally, notice that the last warning doesn't invoke the interpolated string handler. The argument is a `string`, so that call invokes the other overload with a string parameter.

Add more capabilities to the handler

The preceding version of the interpolated string handler implements the pattern. To avoid processing every placeholder expression, you'll need more information in the handler. In this section, you'll improve your handler so that it does less work when the constructed string won't be written to the log. You use [System.Runtime.CompilerServices.InterpolatedStringHandlerArgumentAttribute](#) to specify a mapping between parameters to a public API and parameters to a handler's constructor. That provides the handler with the information needed to determine if the interpolated string should be evaluated.

Let's start with changes to the Handler. First, add a field to track if the handler is enabled. Add two parameters to the constructor: one to specify the log level for this message, and the other a reference to the log object:

```

private readonly bool enabled;

public LogInterpolatedStringHandler(int literalLength, int formattedCount, Logger logger, LogLevel logLevel)
{
    enabled = logger.EnabledLevel >= logLevel;
    builder = new StringBuilder(literalLength);
    Console.WriteLine($"\\tliteral length: {literalLength}, formattedCount: {formattedCount}");
}

```

Next, use the field so that your handler only appends literals or formatted objects when the final string will be used:

```

public void AppendLiteral(string s)
{
    Console.WriteLine($"{tAppendLiteral called: {{{s}}}}");
    if (!enabled) return;

    builder.Append(s);
    Console.WriteLine($"{tAppended the literal string}");
}

public void AppendFormatted<T>(T t)
{
    Console.WriteLine($"{tAppendFormatted called: {{{t}}} is of type {typeof(T)}}");
    if (!enabled) return;

    builder.Append(t?.ToString());
    Console.WriteLine($"{tAppended the formatted object}");
}

```

Next, you'll need to update the `LogMessage` declaration so that the compiler passes the additional parameters to the handler's constructor. That's handled using the [System.Runtime.CompilerServices.InterpolatedStringHandlerArgumentAttribute](#) on the handler argument:

```

public void LogMessage(LogLevel level, [InterpolatedStringHandlerArgument("", "level")]
LogInterpolatedStringHandler builder)
{
    if (EnabledLevel < level) return;
    Console.WriteLine(builder.GetFormattedText());
}

```

This attribute specifies the list of arguments to `LogMessage` that map to the parameters that follow the required `literalLength` and `formattedCount` parameters. The empty string (""), specifies the receiver. The compiler substitutes the value of the `Logger` object represented by `this` for the next argument to the handler's constructor. The compiler substitutes the value of `level` for the following argument. You can provide any number of arguments for any handler you write. The arguments that you add are string arguments.

You can run this version using the same test code. This time, you'll see the following results:

```

    literal length: 65, formattedCount: 1
    AppendLiteral called: {Error Level. CurrentTime: }
    Appended the literal string
    AppendFormatted called: {10/20/2021 12:19:10 PM} is of type System.DateTime
    Appended the formatted object
    AppendLiteral called: {. This is an error. It will be printed.}
    Appended the literal string
Error Level. CurrentTime: 10/20/2021 12:19:10 PM. This is an error. It will be printed.
    literal length: 50, formattedCount: 1
    AppendLiteral called: {Trace Level. CurrentTime: }
    AppendFormatted called: {10/20/2021 12:19:10 PM} is of type System.DateTime
    AppendLiteral called: {. This won't be printed.}
Warning Level. This warning is a string, not an interpolated string expression.

```

You can see that the `AppendLiteral` and `AppendFormat` methods are being called, but they aren't doing any work. The handler has determined that the final string won't be needed, so the handler doesn't build it. There are still a couple of improvements to make.

First, you can add an overload of `AppendFormatted` that constrains the argument to a type that implements [System.IFormattable](#). This overload enables callers to add format strings in the placeholders. While making this change, let's also change the return type of the other `AppendFormatted` and `AppendLiteral` methods, from `void` to `bool` (if any of these methods have different return types, then you'll get a compilation error). That change

enables *short circuiting*. The methods return `false` to indicate that processing of the interpolated string expression should be stopped. Returning `true` indicates that it should continue. In this example, you're using it to stop processing when the resulting string isn't needed. Short circuiting supports more fine-grained actions. You could stop processing the expression once it reaches a certain length, to support fixed-length buffers. Or some condition could indicate remaining elements aren't needed.

```
public void AppendFormatted<T>(T t, string format) where T : IFormattable
{
    Console.WriteLine($"{t}AppendFormatted (IFormattable version) called: {t} with format {{{format}}} is of type {typeof(T)},");

    builder.Append(t?.ToString(format, null));
    Console.WriteLine($"{t}Appended the formatted object");
}
```

With that addition, you can specify format strings in your interpolated string expression:

```
var time = DateTime.Now;

logger.LogMessage(LogLevel.Error, $"Error Level. CurrentTime: {time}. The time doesn't use formatting.");
logger.LogMessage(LogLevel.Error, $"Error Level. CurrentTime: {time:t}. This is an error. It will be printed.");
logger.LogMessage(LogLevel.Trace, $"Trace Level. CurrentTime: {time:t}. This won't be printed.");
```

The `:t` on the first message specifies the "short time format" for the current time. The previous example showed one of the overloads to the `AppendFormatted` method that you can create for your handler. You don't need to specify a generic argument for the object being formatted. You may have more efficient ways to convert types you create to string. You can write overloads of `AppendFormatted` that takes those types instead of a generic argument. The compiler will pick the best overload. The runtime uses this technique to convert `System.Span<T>` to string output. You can add an integer parameter to specify the *alignment* of the output, with or without an `IFormattable`. The `System.Runtime.CompilerServices.DefaultInterpolatedStringHandler` that ships with .NET 6 contains nine overloads of `AppendFormatted` for different uses. You can use it as a reference while building a handler for your purposes.

Run the sample now, and you'll see that for the `Trace` message, only the first `AppendLiteral` is called:

```
literal length: 60, formattedCount: 1
AppendLiteral called: Error Level. CurrentTime:
Appended the literal string
AppendFormatted called: 10/20/2021 12:18:29 PM is of type System.DateTime
Appended the formatted object
AppendLiteral called: . The time doesn't use formatting.
Appended the literal string
Error Level. CurrentTime: 10/20/2021 12:18:29 PM. The time doesn't use formatting.
literal length: 65, formattedCount: 1
AppendLiteral called: Error Level. CurrentTime:
Appended the literal string
AppendFormatted (IFormattable version) called: 10/20/2021 12:18:29 PM with format {t} is of type System.DateTime,
Appended the formatted object
AppendLiteral called: . This is an error. It will be printed.
Appended the literal string
Error Level. CurrentTime: 12:18 PM. This is an error. It will be printed.
literal length: 50, formattedCount: 1
AppendLiteral called: Trace Level. CurrentTime:
Warning Level. This warning is a string, not an interpolated string expression.
```

You can make one final update to the handler's constructor that improves efficiency. The handler can add a final

`out bool` parameter. Setting that parameter to `false` indicates that the handler shouldn't be called at all to process the interpolated string expression:

```
public LogInterpolatedStringHandler(int literalLength, int formattedCount, Logger logger, LogLevel level,
out bool isEnabled)
{
    isEnabled = logger.EnabledLevel >= level;
    Console.WriteLine($"\\tliteral length: {literalLength}, formattedCount: {formattedCount}");
    builder = isEnabled ? new StringBuilder(literalLength) : default!;
}
```

That change means you can remove the `enabled` field. Then, you can change the return type of `AppendLiteral` and `AppendFormatted` to `void`. Now, when you run the sample, you'll see the following output:

```
literal length: 60, formattedCount: 1
AppendLiteral called: Error Level. CurrentTime:
Appended the literal string
AppendFormatted called: 10/20/2021 12:19:10 PM is of type System.DateTime
Appended the formatted object
AppendLiteral called: . The time doesn't use formatting.
Appended the literal string
Error Level. CurrentTime: 10/20/2021 12:19:10 PM. The time doesn't use formatting.
literal length: 65, formattedCount: 1
AppendLiteral called: Error Level. CurrentTime:
Appended the literal string
AppendFormatted (IFormattable version) called: 10/20/2021 12:19:10 PM with format {t} is of type
System.DateTime,
Appended the formatted object
AppendLiteral called: . This is an error. It will be printed.
Appended the literal string
Error Level. CurrentTime: 12:19 PM. This is an error. It will be printed.
literal length: 50, formattedCount: 1
Warning Level. This warning is a string, not an interpolated string expression.
```

The only output when `LogLevel.Trace` was specified is the output from the constructor. The handler indicated that it's not enabled, so none of the `Append` methods were invoked.

This example illustrates an important point for interpolated string handlers, especially when logging libraries are used. Any side-effects in the placeholders may not occur. Add the following code to your main program and see this behavior in action:

```
int index = 0;
int numberOfIncrements = 0;
for (var level = LogLevel.Critical; level <= LogLevel.Trace; level++)
{
    Console.WriteLine(level);
    logger.LogMessage(level, $"{level}: Increment index a few times {index++}, {index++}, {index++},
{index++}, {index++}");
    numberOfIncrements += 5;
}
Console.WriteLine($"Value of index {index}, value of numberOfIncrements: {numberOfIncrements}");
```

You can see the `index` variable is incremented five times each iteration of the loop. Because the placeholders are evaluated only for `Critical`, `Error` and `Warning` levels, not for `Information` and `Trace`, the final value of `index` doesn't match the expectation:

```
Critical
Critical: Increment index a few times 0, 1, 2, 3, 4
Error
Error: Increment index a few times 5, 6, 7, 8, 9
Warning
Warning: Increment index a few times 10, 11, 12, 13, 14
Information
Trace
Value of index 15, value of numberOfIncrements: 25
```

Interpolated string handlers provide greater control over how an interpolated string expression is converted to a string. The .NET runtime team has already used this feature to improve performance in several areas. You can make use of the same capability in your own libraries. To explore further, look at the [System.Runtime.CompilerServices.DefaultInterpolatedStringHandler](#). It provides a more complete implementation than you built here. You'll see many more overloads that are possible for the `Append` methods.

Use string interpolation to construct formatted strings

12/28/2021 • 7 minutes to read • [Edit Online](#)

This tutorial teaches you how to use C# [string interpolation](#) to insert values into a single result string. You write C# code and see the results of compiling and running it. The tutorial contains a series of lessons that show you how to insert values into a string and format those values in different ways.

This tutorial expects that you have a machine you can use for development. The .NET tutorial [Hello World in 10 minutes](#) has instructions for setting up your local development environment on Windows, Linux, or macOS. You can also complete the [interactive version](#) of this tutorial in your browser.

Create an interpolated string

Create a directory named *interpolated*. Make it the current directory and run the following command from a console window:

```
dotnet new console
```

This command creates a new .NET Core console application in the current directory.

Open *Program.cs* in your favorite editor, and replace the line `Console.WriteLine("Hello World!");` with the following code, where you replace `<name>` with your name:

```
var name = "<name>";  
Console.WriteLine($"Hello, {name}. It's a pleasure to meet you!");
```

Try this code by typing `dotnet run` in your console window. When you run the program, it displays a single string that includes your name in the greeting. The string included in the [WriteLine](#) method call is an *interpolated string expression*. It's a kind of template that lets you construct a single string (called the *result string*) from a string that includes embedded code. Interpolated strings are particularly useful for inserting values into a string or concatenating (joining together) strings.

This simple example contains the two elements that every interpolated string must have:

- A string literal that begins with the `$` character before its opening quotation mark character. There can't be any spaces between the `$` symbol and the quotation mark character. (If you'd like to see what happens if you include one, insert a space after the `$` character, save the file, and run the program again by typing `dotnet run` in the console window. The C# compiler displays an error message, "error CS1056: Unexpected character '\$'".)
- One or more *interpolation expressions*. An interpolation expression is indicated by an opening and closing brace (`{` and `}`). You can put any C# expression that returns a value (including `null`) inside the braces.

Let's try a few more string interpolation examples with some other data types.

Include different data types

In the previous section, you used string interpolation to insert one string inside of another. The result of an

interpolation expression can be of any data type, though. Let's include values of various data types in an interpolated string.

In the following example, we first define a `class` data type `Vegetable` that has a `Name` `property` and a `ToString` `method`, which `overrides` the behavior of the `Object.ToString()` method. The `public` `access modifier` makes that method available to any client code to get the string representation of a `Vegetable` instance. In the example the `Vegetable.ToString` method returns the value of the `Name` property that is initialized at the `Vegetable` `constructor`:

```
public Vegetable(string name) => Name = name;
```

Then we create an instance of the `Vegetable` class named `item` by using the `new` `operator` and providing a name for the constructor `Vegetable` :

```
var item = new Vegetable("eggplant");
```

Finally, we include the `item` variable into an interpolated string that also contains a `DateTime` value, a `Decimal` value, and a `Unit` `enumeration` value. Replace all of the C# code in your editor with the following code, and then use the `dotnet run` command to run it:

```
using System;

public class Vegetable
{
    public Vegetable(string name) => Name = name;

    public string Name { get; }

    public override string ToString() => Name;
}

public class Program
{
    public enum Unit { item, kilogram, gram, dozen };

    public static void Main()
    {
        var item = new Vegetable("eggplant");
        var date = DateTime.Now;
        var price = 1.99m;
        var unit = Unit.item;
        Console.WriteLine($"On {date}, the price of {item} was {price} per {unit}.");
    }
}
```

Note that the interpolation expression `item` in the interpolated string resolves to the text "eggplant" in the result string. That's because, when the type of the expression result is not a string, the result is resolved to a string in the following way:

- If the interpolation expression evaluates to `null`, an empty string ("", or `String.Empty`) is used.
- If the interpolation expression doesn't evaluate to `null`, typically the `ToString` method of the result type is called. You can test this by updating the implementation of the `Vegetable.ToString` method. You might not even need to implement the `ToString` method since every type has some implementation of this method. To test this, comment out the definition of the `Vegetable.ToString` method in the example (to do that, put a comment symbol, `//`, in front of it). In the output, the string "eggplant" is replaced by the fully qualified type name ("Vegetable" in this example), which is the default behavior of the `Object.ToString()`

method. The default behavior of the `ToString` method for an enumeration value is to return the string representation of the value.

In the output from this example, the date is too precise (the price of eggplant doesn't change every second), and the price value doesn't indicate a unit of currency. In the next section, you'll learn how to fix those issues by controlling the format of string representations of the expression results.

Control the formatting of interpolation expressions

In the previous section, two poorly formatted strings were inserted into the result string. One was a date and time value for which only the date was appropriate. The second was a price that didn't indicate its unit of currency. Both issues are easy to address. String interpolation lets you specify *format strings* that control the formatting of particular types. Modify the call to `Console.WriteLine` from the previous example to include the format strings for the date and price expressions as shown in the following line:

```
Console.WriteLine($"On {date:d}, the price of {item} was {price:C2} per {unit}.");
```

You specify a format string by following the interpolation expression with a colon (":") and the format string. "d" is a [standard date and time format string](#) that represents the short date format. "C2" is a [standard numeric format string](#) that represents a number as a currency value with two digits after the decimal point.

A number of types in the .NET libraries support a predefined set of format strings. These include all the numeric types and the date and time types. For a complete list of types that support format strings, see [Format Strings and .NET Class Library Types](#) in the [Formatting Types in .NET](#) article.

Try modifying the format strings in your text editor and, each time you make a change, rerun the program to see how the changes affect the formatting of the date and time and the numeric value. Change the "d" in `{date:d}` to "t" (to display the short time format), "y" (to display the year and month), and "yyyy" (to display the year as a four-digit number). Change the "C2" in `{price:C2}` to "e" (for exponential notation) and "F3" (for a numeric value with three digits after the decimal point).

In addition to controlling formatting, you can also control the field width and alignment of the formatted strings that are included in the result string. In the next section, you'll learn how to do this.

Control the field width and alignment of interpolation expressions

Ordinarily, when the result of an interpolation expression is formatted to string, that string is included in a result string without leading or trailing spaces. Particularly when you work with a set of data, being able to control a field width and text alignment helps to produce a more readable output. To see this, replace all the code in your text editor with the following code, then type `dotnet run` to execute the program:

```

using System;
using System.Collections.Generic;

public class Example
{
    public static void Main()
    {
        var titles = new Dictionary<string, string>()
        {
            ["Doyle, Arthur Conan"] = "Hound of the Baskervilles, The",
            ["London, Jack"] = "Call of the Wild, The",
            ["Shakespeare, William"] = "Tempest, The"
        };

        Console.WriteLine("Author and Title List");
        Console.WriteLine();
        Console.WriteLine($"|{"Author",-25}|{"Title",30}|");
        foreach (var title in titles)
            Console.WriteLine($"|{title.Key,-25}|{title.Value,30}|");
    }
}

```

The names of authors are left-aligned, and the titles they wrote are right-aligned. You specify the alignment by adding a comma (",") after an interpolation expression and designating the *minimum* field width. If the specified value is a positive number, the field is right-aligned. If it is a negative number, the field is left-aligned.

Try removing the negative signs from the `{"Author",-25}` and `{title.Key,-25}` code and run the example again, as the following code does:

```

Console.WriteLine($"|{"Author",25}|{"Title",30}|");
foreach (var title in titles)
    Console.WriteLine($"|{title.Key,25}|{title.Value,30}|");

```

This time, the author information is right-aligned.

You can combine an alignment specifier and a format string for a single interpolation expression. To do that, specify the alignment first, followed by a colon and the format string. Replace all of the code inside the `Main` method with the following code, which displays three formatted strings with defined field widths. Then run the program by entering the `dotnet run` command.

```

Console.WriteLine($"[{DateTime.Now,-20:d}] Hour [{DateTime.Now,-10:HH}] [{1063.342,15:N2}] feet");

```

The output looks something like the following:

```

[04/14/2018      ] Hour [16      ] [      1,063.34] feet

```

You've completed the string interpolation tutorial.

For more information, see the [String interpolation](#) topic and the [String interpolation in C#](#) tutorial.

String interpolation in C#

12/28/2021 • 5 minutes to read • [Edit Online](#)

This tutorial shows you how to use [string interpolation](#) to format and include expression results in a result string. The examples assume that you are familiar with basic C# concepts and .NET type formatting. If you are new to string interpolation or .NET type formatting, check out the [interactive string interpolation tutorial](#) first. For more information about formatting types in .NET, see the [Formatting Types in .NET](#) topic.

NOTE

The C# examples in this article run in the [Try.NET](#) inline code runner and playground. Select the **Run** button to run an example in an interactive window. Once you execute the code, you can modify it and run the modified code by selecting **Run** again. The modified code either runs in the interactive window or, if compilation fails, the interactive window displays all C# compiler error messages.

Introduction

The [string interpolation](#) feature is built on top of the [composite formatting](#) feature and provides a more readable and convenient syntax to include formatted expression results in a result string.

To identify a string literal as an interpolated string, prepend it with the `$` symbol. You can embed any valid C# expression that returns a value in an interpolated string. In the following example, as soon as an expression is evaluated, its result is converted into a string and included in a result string:

```
double a = 3;
double b = 4;
Console.WriteLine($"Area of the right triangle with legs of {a} and {b} is {0.5 * a * b}");
Console.WriteLine($"Length of the hypotenuse of the right triangle with legs of {a} and {b} is {CalculateHypotenuse(a, b)}");

double CalculateHypotenuse(double leg1, double leg2) => Math.Sqrt(leg1 * leg1 + leg2 * leg2);

// Expected output:
// Area of the right triangle with legs of 3 and 4 is 6
// Length of the hypotenuse of the right triangle with legs of 3 and 4 is 5
```

As the example shows, you include an expression in an interpolated string by enclosing it with braces:

```
{<interpolationExpression>}
```

Interpolated strings support all the capabilities of the [string composite formatting](#) feature. That makes them a more readable alternative to the use of the [String.Format](#) method.

How to specify a format string for an interpolation expression

You specify a format string that is supported by the type of the expression result by following the interpolation expression with a colon (":") and the format string:

```
{<interpolationExpression>:<formatString>}
```

The following example shows how to specify standard and custom format strings for expressions that produce date and time or numeric results:

```
var date = new DateTime(1731, 11, 25);
Console.WriteLine($"On {date:dddd, MMMM dd, yyyy} Leonhard Euler introduced the letter e to denote
{Math.E:F5} in a letter to Christian Goldbach.");

// Expected output:
// On Sunday, November 25, 1731 Leonhard Euler introduced the letter e to denote 2.71828 in a letter to
Christian Goldbach.
```

For more information, see the [Format String Component](#) section of the [Composite Formatting](#) topic. That section provides links to the topics that describe standard and custom format strings supported by .NET base types.

How to control the field width and alignment of the formatted interpolation expression

You specify the minimum field width and the alignment of the formatted expression result by following the interpolation expression with a comma (",") and the constant expression:

```
{<interpolationExpression>,<alignment>}
```

If the *alignment* value is positive, the formatted expression result is right-aligned; if negative, it's left-aligned.

If you need to specify both alignment and a format string, start with the alignment component:

```
{<interpolationExpression>,<alignment>:<formatString>}
```

The following example shows how to specify alignment and uses pipe characters ("|") to delimit text fields:

```
const int NameAlignment = -9;
const int ValueAlignment = 7;

double a = 3;
double b = 4;
Console.WriteLine($"Three classical Pythagorean means of {a} and {b}:");
Console.WriteLine($"|{"Arithmetic",NameAlignment}|{0.5 * (a + b),ValueAlignment:F3}|");
Console.WriteLine($"|{"Geometric",NameAlignment}|{Math.Sqrt(a * b),ValueAlignment:F3}|");
Console.WriteLine($"|{"Harmonic",NameAlignment}|{2 / (1 / a + 1 / b),ValueAlignment:F3}|");

// Expected output:
// Three classical Pythagorean means of 3 and 4:
// |Arithmetic| 3.500|
// |Geometric| 3.464|
// |Harmonic | 3.429|
```

As the example output shows, if the length of the formatted expression result exceeds specified field width, the *alignment* value is ignored.

For more information, see the [Alignment Component](#) section of the [Composite Formatting](#) topic.

How to use escape sequences in an interpolated string

Interpolated strings support all escape sequences that can be used in ordinary string literals. For more information, see [String escape sequences](#).

To interpret escape sequences literally, use a [verbatim](#) string literal. An interpolated verbatim string starts with the `$` character followed by the `@` character. Starting with C# 8.0, you can use the `$` and `@` tokens in any order: both `$@"..."` and `@$"..."` are valid interpolated verbatim strings.

To include a brace, "{" or "}", in a result string, use two braces, "{{" or "}}". For more information, see the [Escaping Braces](#) section of the [Composite Formatting](#) topic.

The following example shows how to include braces in a result string and construct a verbatim interpolated string:

```
var xs = new int[] { 1, 2, 7, 9 };
var ys = new int[] { 7, 9, 12 };
Console.WriteLine($"Find the intersection of the {{{string.Join(", ",xs)}}} and {{{string.Join(", ",ys)}}} sets.");

var userName = "Jane";
var stringWithEscapes = $"C:\\Users\\{userName}\\Documents";
var verbatimInterpolated = @$"C:\Users\{userName}\Documents";
Console.WriteLine(stringWithEscapes);
Console.WriteLine(verbatimInterpolated);

// Expected output:
// Find the intersection of the {1, 2, 7, 9} and {7, 9, 12} sets.
// C:\Users\Jane\Documents
// C:\Users\Jane\Documents
```

How to use a ternary conditional operator `?:` in an interpolation expression

As the colon (":") has special meaning in an item with an interpolation expression, in order to use a [conditional operator](#) in an expression, enclose it in parentheses, as the following example shows:

```
var rand = new Random();
for (int i = 0; i < 7; i++)
{
    Console.WriteLine($"Coin flip: {(rand.NextDouble() < 0.5 ? "heads" : "tails")}");
}
```

How to create a culture-specific result string with string interpolation

By default, an interpolated string uses the current culture defined by the [CultureInfo.CurrentCulture](#) property for all formatting operations. Use implicit conversion of an interpolated string to a [System.FormattableString](#) instance and call its [ToString\(IFormatProvider\)](#) method to create a culture-specific result string. The following example shows how to do that:

```

var cultures = new System.Globalization.CultureInfo[]
{
    System.Globalization.CultureInfo.GetCultureInfo("en-US"),
    System.Globalization.CultureInfo.GetCultureInfo("en-GB"),
    System.Globalization.CultureInfo.GetCultureInfo("nl-NL"),
    System.Globalization.CultureInfo.InvariantCulture
};

var date = DateTime.Now;
var number = 31_415_926.536;
FormattableString message = $"{date,20}{number,20:N3}";
foreach (var culture in cultures)
{
    var cultureSpecificMessage = message.ToString(culture);
    Console.WriteLine($"{culture.Name,-10}{cultureSpecificMessage}");
}

// Expected output is like:
// en-US      5/17/18 3:44:55 PM      31,415,926.536
// en-GB      17/05/2018 15:44:55      31,415,926.536
// nl-NL      17-05-18 15:44:55      31.415.926,536
//           05/17/2018 15:44:55      31,415,926.536

```

As the example shows, you can use one [FormattableString](#) instance to generate multiple result strings for various cultures.

How to create a result string using the invariant culture

Along with the [FormattableString.ToString\(IFormatProvider\)](#) method, you can use the static [FormattableString.Invariant](#) method to resolve an interpolated string to a result string for the [InvariantCulture](#). The following example shows how to do that:

```

string messageInInvariantCulture = FormattableString.Invariant($"Date and time in invariant culture:
{DateTime.Now}");
Console.WriteLine(messageInInvariantCulture);

// Expected output is like:
// Date and time in invariant culture: 05/17/2018 15:46:24

```

Conclusion

This tutorial describes common scenarios of string interpolation usage. For more information about string interpolation, see the [String interpolation](#) topic. For more information about formatting types in .NET, see the [Formatting Types in .NET](#) and [Composite formatting](#) topics.

See also

- [String.Format](#)
- [System.FormattableString](#)
- [System.IFormattable](#)
- [Strings](#)

Console app

12/28/2021 • 10 minutes to read • [Edit Online](#)

This tutorial teaches you a number of features in .NET and the C# language. You'll learn:

- The basics of the .NET CLI
- The structure of a C# Console Application
- Console I/O
- The basics of File I/O APIs in .NET
- The basics of the Task-based Asynchronous Programming in .NET

You'll build an application that reads a text file, and echoes the contents of that text file to the console. The output to the console is paced to match reading it aloud. You can speed up or slow down the pace by pressing the '<' (less than) or '>' (greater than) keys. You can run this application on Windows, Linux, macOS, or in a Docker container.

There are a lot of features in this tutorial. Let's build them one by one.

Prerequisites

- [.NET 6 SDK](#).
- A code editor.

Create the app

The first step is to create a new application. Open a command prompt and create a new directory for your application. Make that the current directory. Type the command `dotnet new console` at the command prompt. This creates the starter files for a basic "Hello World" application.

Before you start making modifications, let's run the simple Hello World application. After creating the application, type `dotnet run` at the command prompt. This command runs the NuGet package restore process, creates the application executable, and runs the executable.

The simple Hello World application code is all in *Program.cs*. Open that file with your favorite text editor. Replace the code in *Program.cs* with the following code:

```
namespace TeleprompterConsole;

internal class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine("Hello World!");
    }
}
```

At the top of the file, see a `namespace` statement. Like other Object Oriented languages you may have used, C# uses namespaces to organize types. This Hello World program is no different. You can see that the program is in the namespace with the name `TeleprompterConsole`.

Reading and Echoing the File

The first feature to add is the ability to read a text file and display all that text to the console. First, let's add a text file. Copy the [sampleQuotes.txt](#) file from the GitHub repository for this [sample](#) into your project directory. This will serve as the script for your application. For information on how to download the sample app for this tutorial, see the instructions in [Samples and Tutorials](#).

Next, add the following method in your `Program` class (right below the `Main` method):

```
static IEnumerable<string> ReadFrom(string file)
{
    string? line;
    using (var reader = File.OpenText(file))
    {
        while ((line = reader.ReadLine()) != null)
        {
            yield return line;
        }
    }
}
```

This method is a special type of C# method called an *Iterator method*. Enumerator methods return sequences that are evaluated lazily. That means each item in the sequence is generated as it is requested by the code consuming the sequence. Enumerator methods are methods that contain one or more `yield return` statements. The object returned by the `ReadFrom` method contains the code to generate each item in the sequence. In this example, that involves reading the next line of text from the source file, and returning that string. Each time the calling code requests the next item from the sequence, the code reads the next line of text from the file and returns it. When the file is completely read, the sequence indicates that there are no more items.

There are two C# syntax elements that may be new to you. The `using` statement in this method manages resource cleanup. The variable that is initialized in the `using` statement (`reader`, in this example) must implement the `IDisposable` interface. That interface defines a single method, `Dispose`, that should be called when the resource should be released. The compiler generates that call when execution reaches the closing brace of the `using` statement. The compiler-generated code ensures that the resource is released even if an exception is thrown from the code in the block defined by the using statement.

The `reader` variable is defined using the `var` keyword. `var` defines an *implicitly typed local variable*. That means the type of the variable is determined by the compile-time type of the object assigned to the variable. Here, that is the return value from the `OpenText(String)` method, which is a `StreamReader` object.

Now, let's fill in the code to read the file in the `Main` method:

```
var lines = ReadFrom("sampleQuotes.txt");
foreach (var line in lines)
{
    Console.WriteLine(line);
}
```

Run the program (using `dotnet run`) and you can see every line printed out to the console.

Adding Delays and Formatting output

What you have is being displayed far too fast to read aloud. Now you need to add the delays in the output. As you start, you'll be building some of the core code that enables asynchronous processing. However, these first steps will follow a few anti-patterns. The anti-patterns are pointed out in comments as you add the code, and the code will be updated in later steps.

There are two steps to this section. First, you'll update the iterator method to return single words instead of entire lines. That's done with these modifications. Replace the `yield return line;` statement with the following code:

```
var words = line.Split(' ');
foreach (var word in words)
{
    yield return word + " ";
}
yield return Environment.NewLine;
```

Next, you need to modify how you consume the lines of the file, and add a delay after writing each word. Replace the `Console.WriteLine(line)` statement in the `Main` method with the following block:

```
Console.Write(line);
if (!string.IsNullOrEmpty(line))
{
    var pause = Task.Delay(200);
    // Synchronously waiting on a task is an
    // anti-pattern. This will get fixed in later
    // steps.
    pause.Wait();
}
```

Run the sample, and check the output. Now, each single word is printed, followed by a 200 ms delay. However, the displayed output shows some issues because the source text file has several lines that have more than 80 characters without a line break. That can be hard to read while it's scrolling by. That's easy to fix. You'll just keep track of the length of each line, and generate a new line whenever the line length reaches a certain threshold. Declare a local variable after the declaration of `words` in the `ReadFrom` method that holds the line length:

```
var lineLength = 0;
```

Then, add the following code after the `yield return word + " ";` statement (before the closing brace):

```
lineLength += word.Length + 1;
if (lineLength > 70)
{
    yield return Environment.NewLine;
    lineLength = 0;
}
```

Run the sample, and you'll be able to read aloud at its pre-configured pace.

Async Tasks

In this final step, you'll add the code to write the output asynchronously in one task, while also running another task to read input from the user if they want to speed up or slow down the text display, or stop the text display altogether. This has a few steps in it and by the end, you'll have all the updates that you need. The first step is to create an asynchronous [Task](#) returning method that represents the code you've created so far to read and display the file.

Add this method to your `Program` class (it's taken from the body of your `Main` method):

```
private static async Task ShowTeleprompter()
{
    var words = ReadFrom("sampleQuotes.txt");
    foreach (var word in words)
    {
        Console.Write(word);
        if (!string.IsNullOrEmpty(word))
        {
            await Task.Delay(200);
        }
    }
}
```

You'll notice two changes. First, in the body of the method, instead of calling `Wait()` to synchronously wait for a task to finish, this version uses the `await` keyword. In order to do that, you need to add the `async` modifier to the method signature. This method returns a `Task`. Notice that there are no return statements that return a `Task` object. Instead, that `Task` object is created by code the compiler generates when you use the `await` operator. You can imagine that this method returns when it reaches an `await`. The returned `Task` indicates that the work has not completed. The method resumes when the awaited task completes. When it has executed to completion, the returned `Task` indicates that it is complete. Calling code can monitor that returned `Task` to determine when it has completed.

You can call this new method in your `Main` method:

```
ShowTeleprompter().Wait();
```

Here, in `Main`, the code does synchronously wait. You should use the `await` operator instead of synchronously waiting whenever possible. But, in a console application's `Main` method, you cannot use the `await` operator. That would result in the application exiting before all tasks have completed.

NOTE

If you use C# 7.1 or later, you can create console applications with `async Main` method.

Next, you need to write the second asynchronous method to read from the Console and watch for the '<' (less than), '>' (greater than) and 'X' or 'x' keys. Here's the method you add for that task:

```

private static async Task GetInput()
{
    var delay = 200;
    Action work = () =>
    {
        do {
            var key = Console.ReadKey(true);
            if (key.KeyChar == '>')
            {
                delay -= 10;
            }
            else if (key.KeyChar == '<')
            {
                delay += 10;
            }
            else if (key.KeyChar == 'X' || key.KeyChar == 'x')
            {
                break;
            }
        } while (true);
    };
    await Task.Run(work);
}

```

This creates a lambda expression to represent an [Action](#) delegate that reads a key from the Console and modifies a local variable representing the delay when the user presses the '<' (less than) or '>' (greater than) keys. The delegate method finishes when user presses the 'X' or 'x' keys, which allow the user to stop the text display at any time. This method uses [ReadKey\(\)](#) to block and wait for the user to press a key.

To finish this feature, you need to create a new `async Task` returning method that starts both of these tasks (`GetInput` and `ShowTeleprompter`), and also manages the shared data between these two tasks.

It's time to create a class that can handle the shared data between these two tasks. This class contains two public properties: the delay, and a flag `Done` to indicate that the file has been completely read:

```

namespace TeleprompterConsole;

internal class TelePrompterConfig
{
    public int DelayInMilliseconds { get; private set; } = 200;
    public void UpdateDelay(int increment) // negative to speed up
    {
        var newDelay = Min(DelayInMilliseconds + increment, 1000);
        newDelay = Max(newDelay, 20);
        DelayInMilliseconds = newDelay;
    }
    public bool Done { get; private set; }
    public void SetDone()
    {
        Done = true;
    }
}

```

Put that class in a new file, and include that class in the `TeleprompterConsole` namespace as shown. You'll also need to add a `using static` statement at the top of the file so that you can reference the `Min` and `Max` methods without the enclosing class or namespace names. A `using static` statement imports the methods from one class. This is in contrast with the `using` statement without `static`, which imports all classes from a namespace.

```
using static System.Math;
```

Next, you need to update the `ShowTeleprompter` and `GetInput` methods to use the new `config` object. Write one final `Task` returning `async` method to start both tasks and exit when the first task finishes:

```
private static async Task RunTeleprompter()
{
    var config = new TelePrompterConfig();
    var displayTask = ShowTeleprompter(config);

    var speedTask = GetInput(config);
    await Task.WhenAny(displayTask, speedTask);
}
```

The one new method here is the `WhenAny(Task[])` call. That creates a `Task` that finishes as soon as any of the tasks in its argument list completes.

Next, you need to update both the `ShowTeleprompter` and `GetInput` methods to use the `config` object for the delay:

```
private static async Task ShowTeleprompter(TelePrompterConfig config)
{
    var words = ReadFrom("sampleQuotes.txt");
    foreach (var word in words)
    {
        Console.Write(word);
        if (!string.IsNullOrEmpty(word))
        {
            await Task.Delay(config.DelayInMilliseconds);
        }
    }
    config.SetDone();
}

private static async Task GetInput(TelePrompterConfig config)
{
    Action work = () =>
    {
        do {
            var key = Console.ReadKey(true);
            if (key.KeyChar == '>')
                config.UpdateDelay(-10);
            else if (key.KeyChar == '<')
                config.UpdateDelay(10);
            else if (key.KeyChar == 'X' || key.KeyChar == 'x')
                config.SetDone();
        } while (!config.Done);
    };
    await Task.Run(work);
}
```

This new version of `ShowTeleprompter` calls a new method in the `TeleprompterConfig` class. Now, you need to update `Main` to call `RunTeleprompter` instead of `ShowTeleprompter`:

```
RunTeleprompter().Wait();
```

Conclusion

This tutorial showed you a number of the features around the C# language and the .NET Core libraries related to working in Console applications. You can build on this knowledge to explore more about the language, and the classes introduced here. You've seen the basics of File and Console I/O, blocking and non-blocking use of the Task-based asynchronous programming, a tour of the C# language and how C# programs are organized, and the .NET CLI.

For more information about File I/O, see [File and Stream I/O](#). For more information about asynchronous programming model used in this tutorial, see [Task-based Asynchronous Programming](#) and [Asynchronous programming](#).

Tutorial: Make HTTP requests in a .NET console app using C#

12/28/2021 • 8 minutes to read • [Edit Online](#)

This tutorial builds an app that issues HTTP requests to a REST service on GitHub. The app reads information in JSON format and converts the JSON into C# objects. Converting from JSON to C# objects is known as *deserialization*.

The tutorial shows how to:

- Send HTTP requests.
- Deserialize JSON responses.
- Configure deserialization with attributes.

If you prefer to follow along with the [final sample](#) for this tutorial, you can download it. For download instructions, see [Samples and Tutorials](#).

Prerequisites

- [.NET SDK 5.0 or later](#)
- A code editor such as [Visual Studio Code](#), which is an open source, cross platform editor. You can run the sample app on Windows, Linux, or macOS, or in a Docker container.

Create the client app

1. Open a command prompt and create a new directory for your app. Make that the current directory.
2. Enter the following command in a console window:

```
dotnet new console --name WebAPIClient
```

This command creates the starter files for a basic "Hello World" app. The project name is "WebAPIClient".

3. Navigate into the "WebAPIClient" directory, and run the app.

```
cd WebAPIClient
```

```
dotnet run
```

`dotnet run` automatically runs `dotnet restore` to restore any dependencies that the app needs. It also runs `dotnet build` if needed.

Make HTTP requests

This app calls the [GitHub API](#) to get information about the projects under the [.NET Foundation](#) umbrella. The endpoint is <https://api.github.com/orgs/dotnet/repos>. To retrieve information, it makes an HTTP GET request. Browsers also make HTTP GET requests, so you can paste that URL into your browser address bar to see what information you'll be receiving and processing.

Use the [HttpClient](#) class to make HTTP requests. [HttpClient](#) supports only async methods for its long-running APIs. So the following steps create an async method and call it from the Main method.

1. Open the `Program.cs` file in your project directory and add the following async method to the `Program` class:

```
private static async Task ProcessRepositories()
{
}
```

2. Add a `using` directive at the top of the `Program.cs` file so that the C# compiler recognizes the [Task](#) type:

```
using System.Threading.Tasks;
```

If you run `dotnet build` at this point, the compile succeeds but warns that this method doesn't contain any `await` operators and so runs synchronously. You'll add `await` operators later as you fill in the method.

3. Replace the `Main` method with the following code:

```
static async Task Main(string[] args)
{
    await ProcessRepositories();
}
```

This code:

- Changes the signature of `Main` by adding the `async` modifier and changing the return type to `Task`.
- Replaces the `Console.WriteLine` statement with a call to `ProcessRepositories` that uses the `await` keyword.

4. In the `Program` class, create a static instance of [HttpClient](#) to handle requests and responses.

```
namespace WebAPIClient
{
    class Program
    {
        private static readonly HttpClient client = new HttpClient();

        static async Task Main(string[] args)
        {
            //...
        }
    }
}
```

5. In the `ProcessRepositories` method, call the GitHub endpoint that returns a list of all repositories under the .NET foundation organization:

```
private static async Task ProcessRepositories()
{
    client.DefaultRequestHeaders.Accept.Clear();
    client.DefaultRequestHeaders.Accept.Add(
        new MediaTypeWithQualityHeaderValue("application/vnd.github.v3+json"));
    client.DefaultRequestHeaders.Add("User-Agent", ".NET Foundation Repository Reporter");

    var stringTask = client.GetStringAsync("https://api.github.com/orgs/dotnet/repos");

    var msg = await stringTask;
    Console.Write(msg);
}
```

This code:

- Sets up HTTP headers for all requests:
 - An `Accept` header to accept JSON responses
 - A `User-Agent` header. These headers are checked by the GitHub server code and are necessary to retrieve information from GitHub.
- Calls `HttpClient.GetStringAsync(String)` to make a web request and retrieve the response. This method starts a task that makes the web request. When the request returns, the task reads the response stream and extracts the content from the stream. The body of the response is returned as a `String`, which is available when the task completes.
- Awaits the task for the response string and prints the response to the console.

6. Add two `using` directives at the top of the file:

```
using System.Net.Http;
using System.Net.Http.Headers;
```

7. Build the app and run it.

```
dotnet run
```

There is no build warning because the `ProcessRepositories` now contains an `await` operator.

The output is a long display of JSON text.

Deserialize the JSON Result

The following steps convert the JSON response into C# objects. You use the `System.Text.Json.JsonSerializer` class to deserialize JSON into objects.

1. Create a file named *repo.cs* and add the following code:

```
using System;

namespace WebAPIClient
{
    public class Repository
    {
        public string name { get; set; }
    }
}
```

The preceding code defines a class to represent the JSON object returned from the GitHub API. You'll use

this class to display a list of repository names.

The JSON for a repository object contains dozens of properties, but only the `name` property will be deserialized. The serializer automatically ignores JSON properties for which there is no match in the target class. This feature makes it easier to create types that work with only a subset of fields in a large JSON packet.

The C# convention is to [capitalize the first letter of property names](#), but the `name` property here starts with a lowercase letter because that matches exactly what's in the JSON. Later you'll see how to use C# property names that don't match the JSON property names.

2. Use the serializer to convert JSON into C# objects. Replace the call to `GetStringAsync(String)` in the `ProcessRepositories` method with the following lines:

```
var streamTask = client.GetStreamAsync("https://api.github.com/orgs/dotnet/repos");
var repositories = await JsonSerializer.DeserializeAsync<List<Repository>>(await streamTask);
```

The updated code replaces `GetStringAsync(String)` with `GetStreamAsync(String)`. This serializer method uses a stream instead of a string as its source.

The first argument to `JsonSerializer.DeserializeAsync<TValue>(Stream, JsonSerializerOptions, CancellationToken)` is an `await` expression. `await` expressions can appear almost anywhere in your code, even though up to now, you've only seen them as part of an assignment statement. The other two parameters, `JsonSerializerOptions` and `CancellationToken`, are optional and are omitted in the code snippet.

The `DeserializeAsync` method is *generic*, which means you supply type arguments for what kind of objects should be created from the JSON text. In this example, you're deserializing to a `List<Repository>`, which is another generic object, a `System.Collections.Generic.List<T>`. The `List<T>` class stores a collection of objects. The type argument declares the type of objects stored in the `List<T>`. The type argument is your `Repository` class, because the JSON text represents a collection of repository objects.

3. Add code to display the name of each repository. Replace the lines that read:

```
var msg = await stringTask;
Console.Write(msg);
```

with the following code:

```
foreach (var repo in repositories)
    Console.WriteLine(repo.name);
```

4. Add the following `using` directives at the top of the file:

```
using System.Collections.Generic;
using System.Text.Json;
```

5. Run the app.

```
dotnet run
```

The output is a list of the names of the repositories that are part of the .NET Foundation.

Configure deserialization

1. In *repo.cs*, change the `name` property to `Name` and add a `[JsonPropertyName]` attribute to specify how this property appears in the JSON.

```
[JsonPropertyName("name")]
public string Name { get; set; }
```

2. Add the `System.Text.Json.Serialization` namespace to the `using` directives:

```
using System.Text.Json.Serialization;
```

3. In *Program.cs*, update the code to use the new capitalization of the `Name` property:

```
Console.WriteLine(repo.Name);
```

4. Run the app.

The output is the same.

Refactor the code

The `ProcessRepositories` method can do the async work and return a collection of the repositories. Change that method to return `List<Repository>`, and move the code that writes the information into the `Main` method.

1. Change the signature of `ProcessRepositories` to return a task whose result is a list of `Repository` objects:

```
private static async Task<List<Repository>> ProcessRepositories()
```

2. Return the repositories after processing the JSON response:

```
var streamTask = client.GetStreamAsync("https://api.github.com/orgs/dotnet/repos");
var repositories = await JsonSerializer.DeserializeAsync<List<Repository>>(await streamTask);
return repositories;
```

The compiler generates the `Task<T>` object for the return value because you've marked this method as `async`.

3. Modify the `Main` method to capture the results and write each repository name to the console. The `Main` method now looks like this:

```
public static async Task Main(string[] args)
{
    var repositories = await ProcessRepositories();

    foreach (var repo in repositories)
        Console.WriteLine(repo.Name);
}
```

4. Run the app.

The output is the same.

Deserialize more properties

The following steps add code to process more of the properties in the received JSON packet. You probably won't want to process every property, but adding a few more demonstrates other features of C#.

1. Add the following properties to the `Repository` class definition:

```
[JsonPropertyName("description")]
public string Description { get; set; }

[JsonPropertyName("html_url")]
public Uri GitHubHomeUrl { get; set; }

[JsonPropertyName("homepage")]
public Uri Homepage { get; set; }

[JsonPropertyName("watchers")]
public int Watchers { get; set; }
```

The `Uri` and `int` types have built-in functionality to convert to and from string representation. No extra code is needed to deserialize from JSON string format to those target types. If the JSON packet contains data that doesn't convert to a target type, the serialization action throws an exception.

2. Update the `Main` method to display the property values:

```
foreach (var repo in repositories)
{
    Console.WriteLine(repo.Name);
    Console.WriteLine(repo.Description);
    Console.WriteLine(repo.GitHubHomeUrl);
    Console.WriteLine(repo.Homepage);
    Console.WriteLine(repo.Watchers);
    Console.WriteLine();
}
```

3. Run the app.

The list now includes the additional properties.

Add a date property

The date of the last push operation is formatted in this fashion in the JSON response:

```
2016-02-08T21:27:00Z
```

This format is for Coordinated Universal Time (UTC), so the result of deserialization is a `DateTime` value whose `Kind` property is `Utc`.

To get a date and time represented in your time zone, you have to write a custom conversion method.

1. In `repo.cs`, add a `public` property for the UTC representation of the date and time and a `LastPush` `readonly` property that returns the date converted to local time:

```
[JsonPropertyName("pushed_at")]
public DateTime LastPushUtc { get; set; }

public DateTime LastPush => LastPushUtc.ToLocalTime();
```

The `LastPush` property is defined using an *expression-bodied member* for the `get` accessor. There's no `set` accessor. Omitting the `set` accessor is one way to define a *read-only* property in C#. (Yes, you can create *write-only* properties in C#, but their value is limited.)

2. Add another output statement in *Program.cs*: again:

```
Console.WriteLine(repo.LastPush);
```

3. Run the app.

The output includes the date and time of the last push to each repository.

Next steps

In this tutorial, you created an app that makes web requests and parses the results. Your version of the app should now match the [finished sample](#).

Learn more about how to configure JSON serialization in [How to serialize and deserialize \(marshal and unmarshal\) JSON in .NET](#).

Work with Language-Integrated Query (LINQ)

12/28/2021 • 15 minutes to read • [Edit Online](#)

Introduction

This tutorial teaches you features in .NET Core and the C# language. You'll learn how to:

- Generate sequences with LINQ.
- Write methods that can be easily used in LINQ queries.
- Distinguish between eager and lazy evaluation.

You'll learn these techniques by building an application that demonstrates one of the basic skills of any magician: the [faro shuffle](#). Briefly, a faro shuffle is a technique where you split a card deck exactly in half, then the shuffle interleaves each one card from each half to rebuild the original deck.

Magicians use this technique because every card is in a known location after each shuffle, and the order is a repeating pattern.

For your purposes, it is a light hearted look at manipulating sequences of data. The application you'll build constructs a card deck and then performs a sequence of shuffles, writing the sequence out each time. You'll also compare the updated order to the original order.

This tutorial has multiple steps. After each step, you can run the application and see the progress. You can also see the [completed sample](#) in the dotnet/samples GitHub repository. For download instructions, see [Samples and Tutorials](#).

Prerequisites

You'll need to set up your machine to run .NET core. You can find the installation instructions on the [.NET Core Download](#) page. You can run this application on Windows, Ubuntu Linux, or OS X, or in a Docker container. You'll need to install your favorite code editor. The descriptions below use [Visual Studio Code](#) which is an open source, cross-platform editor. However, you can use whatever tools you are comfortable with.

Create the Application

The first step is to create a new application. Open a command prompt and create a new directory for your application. Make that the current directory. Type the command `dotnet new console` at the command prompt. This creates the starter files for a basic "Hello World" application.

If you've never used C# before, [this tutorial](#) explains the structure of a C# program. You can read that and then return here to learn more about LINQ.

Create the Data Set

Before you begin, make sure that the following lines are at the top of the `Program.cs` file generated by

```
dotnet new console :
```

```
// Program.cs
using System;
using System.Collections.Generic;
using System.Linq;
```

If these three lines (`using` statements) aren't at the top of the file, our program will not compile.

Now that you have all of the references that you'll need, consider what constitutes a deck of cards. Commonly, a deck of playing cards has four suits, and each suit has thirteen values. Normally, you might consider creating a `Card` class right off the bat and populating a collection of `Card` objects by hand. With LINQ, you can be more concise than the usual way of dealing with creating a deck of cards. Instead of creating a `Card` class, you can create two sequences to represent suits and ranks, respectively. You'll create a really simple pair of *iterator methods* that will generate the ranks and suits as `IEnumerable<T>`s of strings:

```
// Program.cs
// The Main() method

static IEnumerable<string> Suits()
{
    yield return "clubs";
    yield return "diamonds";
    yield return "hearts";
    yield return "spades";
}

static IEnumerable<string> Ranks()
{
    yield return "two";
    yield return "three";
    yield return "four";
    yield return "five";
    yield return "six";
    yield return "seven";
    yield return "eight";
    yield return "nine";
    yield return "ten";
    yield return "jack";
    yield return "queen";
    yield return "king";
    yield return "ace";
}
```

Place these underneath the `Main` method in your `Program.cs` file. These two methods both utilize the `yield return` syntax to produce a sequence as they run. The compiler builds an object that implements `IEnumerable<T>` and generates the sequence of strings as they are requested.

Now, use these iterator methods to create the deck of cards. You'll place the LINQ query in our `Main` method. Here's a look at it:

```
// Program.cs
static void Main(string[] args)
{
    var startingDeck = from s in Suits()
                      from r in Ranks()
                      select new { Suit = s, Rank = r };

    // Display each card that we've generated and placed in startingDeck in the console
    foreach (var card in startingDeck)
    {
        Console.WriteLine(card);
    }
}
```

The multiple `from` clauses produce a `SelectMany`, which creates a single sequence from combining each element in the first sequence with each element in the second sequence. The order is important for our purposes. The first element in the first source sequence (Suits) is combined with every element in the second

sequence (Ranks). This produces all thirteen cards of first suit. That process is repeated with each element in the first sequence (Suits). The end result is a deck of cards ordered by suits, followed by values.

It's important to keep in mind that whether you choose to write your LINQ in the query syntax used above or use method syntax instead, it's always possible to go from one form of syntax to the other. The above query written in query syntax can be written in method syntax as:

```
var startingDeck = Suits().SelectMany(suit => Ranks().Select(rank => new { Suit = suit, Rank = rank }));
```

The compiler translates LINQ statements written with query syntax into the equivalent method call syntax. Therefore, regardless of your syntax choice, the two versions of the query produce the same result. Choose which syntax works best for your situation: for instance, if you're working in a team where some of the members have difficulty with method syntax, try to prefer using query syntax.

Go ahead and run the sample you've built at this point. It will display all 52 cards in the deck. You may find it very helpful to run this sample under a debugger to observe how the `Suits()` and `Ranks()` methods execute. You can clearly see that each string in each sequence is generated only as it is needed.

```
C:\Windows\system32\cmd.exe
C:\console-linq>dotnet run
{
  Suit = clubs, Rank = two }
  Suit = clubs, Rank = three }
  Suit = clubs, Rank = four }
  Suit = clubs, Rank = five }
  Suit = clubs, Rank = six }
  Suit = clubs, Rank = seven }
  Suit = clubs, Rank = eight }
  Suit = clubs, Rank = nine }
  Suit = clubs, Rank = ten }
  Suit = clubs, Rank = jack }
  Suit = clubs, Rank = queen }
  Suit = clubs, Rank = king }
  Suit = clubs, Rank = ace }
  Suit = diamonds, Rank = two }
  Suit = diamonds, Rank = three }
  Suit = diamonds, Rank = four }
  Suit = diamonds, Rank = five }
  Suit = diamonds, Rank = six }
  Suit = diamonds, Rank = seven }
  Suit = diamonds, Rank = eight }
  Suit = diamonds, Rank = nine }
  Suit = diamonds, Rank = ten }
```

Manipulate the Order

Next, focus on how you're going to shuffle the cards in the deck. The first step in any good shuffle is to split the deck in two. The `Take` and `Skip` methods that are part of the LINQ APIs provide that feature for you. Place them underneath the `foreach` loop:

```
// Program.cs
public static void Main(string[] args)
{
    var startingDeck = from s in Suits()
                       from r in Ranks()
                       select new { Suit = s, Rank = r };

    foreach (var c in startingDeck)
    {
        Console.WriteLine(c);
    }

    // 52 cards in a deck, so 52 / 2 = 26
    var top = startingDeck.Take(26);
    var bottom = startingDeck.Skip(26);
}
```

However, there's no shuffle method to take advantage of in the standard library, so you'll have to write your own. The shuffle method you'll be creating illustrates several techniques that you'll use with LINQ-based programs, so each part of this process will be explained in steps.

In order to add some functionality to how you interact with the `IEnumerable<T>` you'll get back from LINQ queries, you'll need to write some special kinds of methods called [extension methods](#). Briefly, an extension method is a special purpose *static method* that adds new functionality to an already-existing type without having to modify the original type you want to add functionality to.

Give your extension methods a new home by adding a new *static* class file to your program called

`Extensions.cs`, and then start building out the first extension method:

```
// Extensions.cs
using System;
using System.Collections.Generic;
using System.Linq;

namespace LinqFaroShuffle
{
    public static class Extensions
    {
        public static IEnumerable<T> InterleaveSequenceWith<T>(this IEnumerable<T> first, IEnumerable<T>
second)
        {
            // Your implementation will go here soon enough
        }
    }
}
```

Look at the method signature for a moment, specifically the parameters:

```
public static IEnumerable<T> InterleaveSequenceWith<T> (this IEnumerable<T> first, IEnumerable<T> second)
```

You can see the addition of the `this` modifier on the first argument to the method. That means you call the method as though it were a member method of the type of the first argument. This method declaration also follows a standard idiom where the input and output types are `IEnumerable<T>`. That practice enables LINQ methods to be chained together to perform more complex queries.

Naturally, since you split the deck into halves, you'll need to join those halves together. In code, this means you'll be enumerating both of the sequences you acquired through [Take](#) and [Skip](#) at once, [interleaving](#) the elements, and creating one sequence: your now-shuffled deck of cards. Writing a LINQ method that works with two sequences requires that you understand how `IEnumerable<T>` works.

The `IEnumerable<T>` interface has one method: [GetEnumerator](#). The object returned by [GetEnumerator](#) has a method to move to the next element, and a property that retrieves the current element in the sequence. You will use those two members to enumerate the collection and return the elements. This Interleave method will be an iterator method, so instead of building a collection and returning the collection, you'll use the `yield return` syntax shown above.

Here's the implementation of that method:


```

public static IEnumerable<T> InterleaveSequenceWith<T>
    (this IEnumerable<T> first, IEnumerable<T> second)
{
    var firstIter = first.GetEnumerator();
    var secondIter = second.GetEnumerator();

    while (firstIter.MoveNext() && secondIter.MoveNext())
    {
        yield return firstIter.Current;
        yield return secondIter.Current;
    }
}

```

Now that you've written this method, go back to the `Main` method and shuffle the deck once:

```

// Program.cs
public static void Main(string[] args)
{
    var startingDeck = from s in Suits()
                       from r in Ranks()
                       select new { Suit = s, Rank = r };

    foreach (var c in startingDeck)
    {
        Console.WriteLine(c);
    }

    var top = startingDeck.Take(26);
    var bottom = startingDeck.Skip(26);
    var shuffle = top.InterleaveSequenceWith(bottom);

    foreach (var c in shuffle)
    {
        Console.WriteLine(c);
    }
}

```

Comparisons

How many shuffles it takes to set the deck back to its original order? To find out, you'll need to write a method that determines if two sequences are equal. After you have that method, you'll need to place the code that shuffles the deck in a loop, and check to see when the deck is back in order.

Writing a method to determine if the two sequences are equal should be straightforward. It's a similar structure to the method you wrote to shuffle the deck. Only this time, instead of `yield return`ing each element, you'll compare the matching elements of each sequence. When the entire sequence has been enumerated, if every element matches, the sequences are the same:

```

public static bool SequenceEquals<T>
    (this IEnumerable<T> first, IEnumerable<T> second)
{
    var firstIter = first.GetEnumerator();
    var secondIter = second.GetEnumerator();

    while (firstIter.MoveNext() && secondIter.MoveNext())
    {
        if (!firstIter.Current.Equals(secondIter.Current))
        {
            return false;
        }
    }

    return true;
}

```

This shows a second LINQ idiom: terminal methods. They take a sequence as input (or in this case, two sequences), and return a single scalar value. When using terminal methods, they are always the final method in a chain of methods for a LINQ query, hence the name "terminal".

You can see this in action when you use it to determine when the deck is back in its original order. Put the shuffle code inside a loop, and stop when the sequence is back in its original order by applying the `SequenceEquals()` method. You can see it would always be the final method in any query, because it returns a single value instead of a sequence:

```

// Program.cs
static void Main(string[] args)
{
    // Query for building the deck

    // Shuffling using InterleaveSequenceWith<T>();

    var times = 0;
    // We can re-use the shuffle variable from earlier, or you can make a new one
    shuffle = startingDeck;
    do
    {
        shuffle = shuffle.Take(26).InterleaveSequenceWith(shuffle.Skip(26));

        foreach (var card in shuffle)
        {
            Console.WriteLine(card);
        }
        Console.WriteLine();
        times++;
    } while (!startingDeck.SequenceEquals(shuffle));

    Console.WriteLine(times);
}

```

Run the code you've got so far and take note of how the deck rearranges on each shuffle. After 8 shuffles (iterations of the do-while loop), the deck returns to the original configuration it was in when you first created it from the starting LINQ query.

Optimizations

The sample you've built so far executes an *out shuffle*, where the top and bottom cards stay the same on each run. Let's make one change: we'll use an *in shuffle* instead, where all 52 cards change position. For an in shuffle, you interleave the deck so that the first card in the bottom half becomes the first card in the deck. That means

the last card in the top half becomes the bottom card. This is a simple change to a singular line of code. Update the current shuffle query by switching the positions of [Take](#) and [Skip](#). This will change the order of the top and bottom halves of the deck:

```
shuffle = shuffle.Skip(26).InterleaveSequenceWith(shuffle.Take(26));
```

Run the program again, and you'll see that it takes 52 iterations for the deck to reorder itself. You'll also start to notice some serious performance degradations as the program continues to run.

There are a number of reasons for this. You can tackle one of the major causes of this performance drop: inefficient use of [lazy evaluation](#).

Briefly, lazy evaluation states that the evaluation of a statement is not performed until its value is needed. LINQ queries are statements that are evaluated lazily. The sequences are generated only as the elements are requested. Usually, that's a major benefit of LINQ. However, in a use such as this program, this causes exponential growth in execution time.

Remember that we generated the original deck using a LINQ query. Each shuffle is generated by performing three LINQ queries on the previous deck. All these are performed lazily. That also means they are performed again each time the sequence is requested. By the time you get to the 52nd iteration, you're regenerating the original deck many, many times. Let's write a log to demonstrate this behavior. Then, you'll fix it.

In your `Extensions.cs` file, type in or copy the method below. This extension method creates a new file called `debug.log` within your project directory and records what query is currently being executed to the log file. This extension method can be appended to any query to mark that the query executed.

```
public static IEnumerable<T> LogQuery<T>
    (this IEnumerable<T> sequence, string tag)
{
    // File.AppendText creates a new file if the file doesn't exist.
    using (var writer = File.AppendText("debug.log"))
    {
        writer.WriteLine($"Executing Query {tag}");
    }

    return sequence;
}
```

You will see a red squiggle under `File`, meaning it doesn't exist. It won't compile, since the compiler doesn't know what `File` is. To solve this problem, make sure to add the following line of code under the very first line in `Extensions.cs`:

```
using System.IO;
```

This should solve the issue and the red error disappears.

Next, instrument the definition of each query with a log message:

```
// Program.cs
public static void Main(string[] args)
{
    var startingDeck = (from s in Suits().LogQuery("Suit Generation")
                        from r in Ranks().LogQuery("Rank Generation")
                        select new { Suit = s, Rank = r }).LogQuery("Starting Deck");

    foreach (var c in startingDeck)
    {
        Console.WriteLine(c);
    }

    Console.WriteLine();
    var times = 0;
    var shuffle = startingDeck;

    do
    {
        // Out shuffle
        /*
        shuffle = shuffle.Take(26)
            .LogQuery("Top Half")
            .InterleaveSequenceWith(shuffle.Skip(26)
            .LogQuery("Bottom Half"))
            .LogQuery("Shuffle");
        */

        // In shuffle
        shuffle = shuffle.Skip(26).LogQuery("Bottom Half")
            .InterleaveSequenceWith(shuffle.Take(26).LogQuery("Top Half"))
            .LogQuery("Shuffle");

        foreach (var c in shuffle)
        {
            Console.WriteLine(c);
        }

        times++;
        Console.WriteLine(times);
    } while (!startingDeck.SequenceEquals(shuffle));

    Console.WriteLine(times);
}
```

Notice that you don't log every time you access a query. You log only when you create the original query. The program still takes a long time to run, but now you can see why. If you run out of patience running the in shuffle with logging turned on, switch back to the out shuffle. You'll still see the lazy evaluation effects. In one run, it executes 2592 queries, including all the value and suit generation.

You can improve the performance of the code here to reduce the number of executions you make. A simple fix you can make is to *cache* the results of the original LINQ query that constructs the deck of cards. Currently, you're executing the queries again and again every time the do-while loop goes through an iteration, re-constructing the deck of cards and reshuffling it every time. To cache the deck of cards, you can leverage the LINQ methods [ToArray](#) and [ToList](#); when you append them to the queries, they'll perform the same actions you've told them to, but now they'll store the results in an array or a list, depending on which method you choose to call. Append the LINQ method [ToArray](#) to both queries and run the program again:

```

public static void Main(string[] args)
{
    var startingDeck = (from s in Suits().LogQuery("Suit Generation")
                        from r in Ranks().LogQuery("Value Generation")
                        select new { Suit = s, Rank = r })
                        .LogQuery("Starting Deck")
                        .ToArray();

    foreach (var c in startingDeck)
    {
        Console.WriteLine(c);
    }

    Console.WriteLine();

    var times = 0;
    var shuffle = startingDeck;

    do
    {
        /*
        shuffle = shuffle.Take(26)
            .LogQuery("Top Half")
            .InterleaveSequenceWith(shuffle.Skip(26).LogQuery("Bottom Half"))
            .LogQuery("Shuffle")
            .ToArray();
        */

        shuffle = shuffle.Skip(26)
            .LogQuery("Bottom Half")
            .InterleaveSequenceWith(shuffle.Take(26).LogQuery("Top Half"))
            .LogQuery("Shuffle")
            .ToArray();

        foreach (var c in shuffle)
        {
            Console.WriteLine(c);
        }

        times++;
        Console.WriteLine(times);
    } while (!startingDeck.SequenceEquals(shuffle));

    Console.WriteLine(times);
}

```

Now the out shuffle is down to 30 queries. Run again with the in shuffle and you'll see similar improvements: it now executes 162 queries.

Please note that this example is **designed** to highlight the use cases where lazy evaluation can cause performance difficulties. While it's important to see where lazy evaluation can impact code performance, it's equally important to understand that not all queries should run eagerly. The performance hit you incur without using [ToArray](#) is because each new arrangement of the deck of cards is built from the previous arrangement. Using lazy evaluation means each new deck configuration is built from the original deck, even executing the code that built the `startingDeck`. That causes a large amount of extra work.

In practice, some algorithms run well using eager evaluation, and others run well using lazy evaluation. For daily usage, lazy evaluation is usually a better choice when the data source is a separate process, like a database engine. For databases, lazy evaluation allows more complex queries to execute only one round trip to the database process and back to the rest of your code. LINQ is flexible whether you choose to utilize lazy or eager evaluation, so measure your processes and pick whichever kind of evaluation gives you the best performance.

Conclusion

In this project, you covered:

- using LINQ queries to aggregate data into a meaningful sequence
- writing Extension methods to add our own custom functionality to LINQ queries
- locating areas in our code where our LINQ queries might run into performance issues like degraded speed
- lazy and eager evaluation in regards to LINQ queries and the implications they might have on query performance

Aside from LINQ, you learned a bit about a technique magicians use for card tricks. Magicians use the Faro shuffle because they can control where every card moves in the deck. Now that you know, don't spoil it for everyone else!

For more information on LINQ, see:

- [Language Integrated Query \(LINQ\)](#)
- [Introduction to LINQ](#)
- [Basic LINQ Query Operations \(C#\)](#)
- [Data Transformations With LINQ \(C#\)](#)
- [Query Syntax and Method Syntax in LINQ \(C#\)](#)
- [C# Features That Support LINQ](#)

Use Attributes in C#

12/28/2021 • 7 minutes to read • [Edit Online](#)

Attributes provide a way of associating information with code in a declarative way. They can also provide a reusable element that can be applied to a variety of targets.

Consider the `[Obsolete]` attribute. It can be applied to classes, structs, methods, constructors, and more. It *declares* that the element is obsolete. It's then up to the C# compiler to look for this attribute, and do some action in response.

In this tutorial, you'll be introduced to how to add attributes to your code, how to create and use your own attributes, and how to use some attributes that are built into .NET Core.

Prerequisites

You'll need to set up your machine to run .NET core. You can find the installation instructions on the [.NET Core Downloads](#) page. You can run this application on Windows, Ubuntu Linux, macOS or in a Docker container. You'll need to install your favorite code editor. The descriptions below use [Visual Studio Code](#) which is an open source, cross platform editor. However, you can use whatever tools you are comfortable with.

Create the Application

Now that you've installed all the tools, create a new .NET Core application. To use the command line generator, execute the following command in your favorite shell:

```
dotnet new console
```

This command will create bare-bones .NET core project files. You will need to execute `dotnet restore` to restore the dependencies needed to compile this project.

You don't have to run `dotnet restore` because it's run implicitly by all commands that require a restore to occur, such as `dotnet new`, `dotnet build`, `dotnet run`, `dotnet test`, `dotnet publish`, and `dotnet pack`. To disable implicit restore, use the `--no-restore` option.

The `dotnet restore` command is still useful in certain scenarios where explicitly restoring makes sense, such as [continuous integration builds in Azure DevOps Services](#) or in build systems that need to explicitly control when the restore occurs.

For information about how to manage NuGet feeds, see the [dotnet restore documentation](#).

To execute the program, use `dotnet run`. You should see "Hello, World" output to the console.

How to add attributes to code

In C#, attributes are classes that inherit from the `Attribute` base class. Any class that inherits from `Attribute` can be used as a sort of "tag" on other pieces of code. For instance, there is an attribute called `ObsoleteAttribute`. This is used to signal that code is obsolete and shouldn't be used anymore. You can place this attribute on a class, for instance, by using square brackets.

```
[Obsolete]
public class MyClass
{
}
```

Note that while the class is called `ObsoleteAttribute`, it's only necessary to use `[Obsolete]` in the code. This is a convention that C# follows. You can use the full name `[ObsoleteAttribute]` if you choose.

When marking a class obsolete, it's a good idea to provide some information as to *why* it's obsolete, and/or *what* to use instead. Do this by passing a string parameter to the `Obsolete` attribute.

```
[Obsolete("ThisClass is obsolete. Use ThisClass2 instead.")]
public class ThisClass
{
}
```

The string is being passed as an argument to an `ObsoleteAttribute` constructor, just as if you were writing `var attr = new ObsoleteAttribute("some string")`.

Parameters to an attribute constructor are limited to simple types/literals:

`bool`, `int`, `double`, `string`, `Type`, `enums`, `etc` and arrays of those types. You can not use an expression or a variable. You are free to use positional or named parameters.

How to create your own attribute

Creating an attribute is as simple as inheriting from the `Attribute` base class.

```
public class MySpecialAttribute : Attribute
{
}
```

With the above, I can now use `[MySpecial]` (or `[MySpecialAttribute]`) as an attribute elsewhere in the code base.

```
[MySpecial]
public class SomeOtherClass
{
}
```

Attributes in the .NET base class library like `ObsoleteAttribute` trigger certain behaviors within the compiler. However, any attribute you create acts only as metadata, and doesn't result in any code within the attribute class being executed. It's up to you to act on that metadata elsewhere in your code (more on that later in the tutorial).

There is a 'gotcha' here to watch out for. As mentioned above, only certain types are allowed to be passed as arguments when using attributes. However, when creating an attribute type, the C# compiler won't stop you from creating those parameters. In the below example, I've created an attribute with a constructor that compiles just fine.

```
public class GotchaAttribute : Attribute
{
    public GotchaAttribute(Foo myClass, string str) {
    }
}
```


However, you will be unable to use this constructor with attribute syntax.

```
[Gotcha(new Foo(), "test")] // does not compile
public class AttributeFail
{
}
```

The above will cause a compiler error like

```
Attribute constructor parameter 'myClass' has type 'Foo', which is not a valid attribute parameter type
```

How to restrict attribute usage

Attributes can be used on a number of "targets". The above examples show them on classes, but they can also be used on:

- Assembly
- Class
- Constructor
- Delegate
- Enum
- Event
- Field
- GenericParameter
- Interface
- Method
- Module
- Parameter
- Property
- ReturnValue
- Struct

When you create an attribute class, by default, C# will allow you to use that attribute on any of the possible attribute targets. If you want to restrict your attribute to certain targets, you can do so by using the

`AttributeUsageAttribute` on your attribute class. That's right, an attribute on an attribute!

```
[AttributeUsage(AttributeTargets.Class | AttributeTargets.Struct)]
public class MyAttributeForClassAndStructOnly : Attribute
{
}
```

If you attempt to put the above attribute on something that's not a class or a struct, you will get a compiler error like

```
Attribute 'MyAttributeForClassAndStructOnly' is not valid on this declaration type. It is only valid on
'class, struct' declarations
```

```
public class Foo
{
    // if the below attribute was uncommented, it would cause a compiler error
    // [MyAttributeForClassAndStructOnly]
    public Foo()
    { }
}
```

How to use attributes attached to a code element

Attributes act as metadata. Without some outward force, they won't actually do anything.

To find and act on attributes, [Reflection](#) is generally needed. I won't cover Reflection in-depth in this tutorial, but the basic idea is that Reflection allows you to write code in C# that examines other code.

For instance, you can use Reflection to get information about a class (add `using System.Reflection;` at the head of your code):

```
TypeInfo typeInfo = typeof(MyClass).GetTypeInfo();
Console.WriteLine("The assembly qualified name of MyClass is " + typeInfo.AssemblyQualifiedName);
```

That will print out something like:

```
The assembly qualified name of MyClass is ConsoleApplication.MyClass, attributes, Version=1.0.0.0,
Culture=neutral, PublicKeyToken=null
```

Once you have a `TypeInfo` object (or a `MemberInfo`, `FieldInfo`, etc), you can use the `GetCustomAttributes` method. This will return a collection of `Attribute` objects. You can also use `GetCustomAttribute` and specify an Attribute type.

Here's an example of using `GetCustomAttributes` on a `MemberInfo` instance for `MyClass` (which we saw earlier has an `[Obsolete]` attribute on it).

```
var attrs = typeInfo.GetCustomAttributes();
foreach(var attr in attrs)
    Console.WriteLine("Attribute on MyClass: " + attr.GetType().Name);
```

That will print to console: `Attribute on MyClass: ObsoleteAttribute`. Try adding other attributes to `MyClass`.

It's important to note that these `Attribute` objects are instantiated lazily. That is, they won't be instantiated until you use `GetCustomAttribute` or `GetCustomAttributes`. They are also instantiated each time. Calling `GetCustomAttributes` twice in a row will return two different instances of `ObsoleteAttribute`.

Common attributes in the base class library (BCL)

Attributes are used by many tools and frameworks. NUnit uses attributes like `[Test]` and `[TestFixture]` that are used by the NUnit test runner. ASPNET MVC uses attributes like `[Authorize]` and provides an action filter framework to perform cross-cutting concerns on MVC actions. [PostSharp](#) uses the attribute syntax to allow aspect-oriented programming in C#.

Here are a few notable attributes built into the .NET Core base class libraries:

- `[Obsolete]`. This one was used in the above examples, and it lives in the `System` namespace. It is useful to provide declarative documentation about a changing code base. A message can be provided in the form of a string, and another boolean parameter can be used to escalate from a compiler warning to a compiler error.
- `[Conditional]`. This attribute is in the `System.Diagnostics` namespace. This attribute can be applied to methods (or attribute classes). You must pass a string to the constructor. If that string doesn't match a `#define` directive, then any calls to that method (but not the method itself) will be removed by the C# compiler. Typically this is used for debugging (diagnostics) purposes.
- `[CallerMemberName]`. This attribute can be used on parameters, and lives in the `System.Runtime.CompilerServices` namespace. This is an attribute that is used to inject the name of the method that is calling another method. This is typically used as a way to eliminate 'magic strings' when

implementing `INotifyPropertyChanged` in various UI frameworks. As an example:

```
public class MyUIClass : INotifyPropertyChanged
{
    public event PropertyChangedEventHandler PropertyChanged;

    public void RaisePropertyChanged([CallerMemberName] string propertyName = null)
    {
        PropertyChanged?.Invoke(this, new PropertyChangedEventArgs(propertyName));
    }

    private string _name;
    public string Name
    {
        get { return _name; }
        set
        {
            if (value != _name)
            {
                _name = value;
                RaisePropertyChanged(); // notice that "Name" is not needed here explicitly
            }
        }
    }
}
```

In the above code, you don't have to have a literal `"Name"` string. This can help prevent typo-related bugs and also makes for smoother refactoring/renaming.

Summary

Attributes bring declarative power to C#, but they are a meta-data form of code and don't act by themselves.

Nullable reference types

12/28/2021 • 15 minutes to read • [Edit Online](#)

Prior to C# 8.0, all reference types were nullable. *Nullable reference types* refers to a group of features introduced in C# 8.0 that you can use to minimize the likelihood that your code causes the runtime to throw [System.NullReferenceException](#). *Nullable reference types* includes three features that help you avoid these exceptions, including the ability to explicitly mark a reference type as *nullable*:

- Improved static flow analysis that determines if a variable may be `null` before dereferencing it.
- Attributes that annotate APIs so that the flow analysis determines *null-state*.
- Variable annotations that developers use to explicitly declare the intended *null-state* for a variable.

Null-state analysis and variable annotations are disabled by default for existing projects—meaning that all reference types continue to be nullable. Starting in .NET 6, they're enabled by default for *new* projects. For information about enabling these features by declaring a *nullable annotation context*, see [Nullable contexts](#).

The rest of this article describes how those three feature areas work to produce warnings when your code may be **dereferencing** a `null` value. Dereferencing a variable means to access one of its members using the `.` (dot) operator, as shown in the following example:

```
string message = "Hello, World!";  
int length = message.Length; // dereferencing "message"
```

When you dereference a variable whose value is `null`, the runtime throws a [System.NullReferenceException](#).

Null state analysis

Null-state analysis tracks the *null-state* of references. This static analysis emits warnings when your code may dereference `null`. You can address these warnings to minimize incidences when the runtime throws a [System.NullReferenceException](#). The compiler uses static analysis to determine the *null-state* of a variable. A variable is either *not-null* or *maybe-null*. The compiler determines that a variable is *not-null* in two ways:

1. The variable has been assigned to a value that is known to be *not null*.
2. The variable has been checked against `null` and hasn't been modified since that check.

Any variable that the compiler hasn't determined as *not-null* is considered *maybe-null*. The analysis provides warnings in situations where you may accidentally dereference a `null` value. The compiler produces warnings based on the *null-state*.

- When a variable is *not-null*, that variable may be dereferenced safely.
- When a variable is *maybe-null*, that variable must be checked to ensure that it isn't `null` before dereferencing it.

Consider the following example:

```

string message = null;

// warning: dereference null.
Console.WriteLine($"The length of the message is {message.Length}");

var originalMessage = message;
message = "Hello, World!";

// No warning. Analysis determined "message" is not null.
Console.WriteLine($"The length of the message is {message.Length}");

// warning!
Console.WriteLine(originalMessage.Length);

```

In the preceding example, the compiler determines that `message` is *maybe-null* when the first message is printed. There's no warning for the second message. The final line of code produces a warning because `originalMessage` might be null. The following example shows a more practical use for traversing a tree of nodes to the root, processing each node during the traversal:

```

void FindRoot(Node node, Action<Node> processNode)
{
    for (var current = node; current != null; current = current.Parent)
    {
        processNode(current);
    }
}

```

The previous code doesn't generate any warnings for dereferencing the variable `current`. Static analysis determines that `current` is never dereferenced when it's *maybe-null*. The variable `current` is checked against `null` before `current.Parent` is accessed, and before passing `current` to the `ProcessNode` action. The previous examples show how the compiler determines *null-state* for local variables when initialized, assigned, or compared to `null`.

NOTE

A number of improvements to definite assignment and null state analysis were added in C# 10. When you upgrade to C# 10, you may find fewer nullable warnings that are false positives. You can learn more about the improvements in the [features specification for definite assignment improvements](#).

Attributes on API signatures

The null state analysis needs hints from developers to understand the semantics of APIs. Some APIs provide null checks, and should change the *null-state* of a variable from *maybe-null* to *not-null*. Other APIs return expressions that are *not-null* or *maybe-null* depending on the *null-state* of the input arguments. For example, consider the following code that displays a message:

```

public void PrintMessage(string message)
{
    if (!string.IsNullOrEmpty(message))
    {
        Console.WriteLine($"{DateTime.Now}: {message}");
    }
}

```

Based on inspection, any developer would consider this code safe, and shouldn't generate warnings. The compiler doesn't know that `IsNullOrEmpty` provides a null check. You apply attributes to inform the

compiler that `message` is *not-null* if and only if `IsNullOrWhiteSpace` returns `false`. In the previous example, the signature includes the `NotNullWhen` to indicate the null state of `message`:

```
public static bool IsNullOrWhiteSpace([NotNullWhen(false)] string message);
```

Attributes provide detailed information about the null state of arguments, return values, and members of the object instance used to invoke a member. The details on each attribute can be found in the language reference article on [nullable reference attributes](#). The .NET runtime APIs have all been annotated in .NET 5. You improve the static analysis by annotating your APIs to provide semantic information about the *null-state* of arguments and return values.

Nullable variable annotations

The *null-state* analysis provides robust analysis for most variables. The compiler needs more information from you for member variables. The compiler can't make assumptions about the order in which public members are accessed. Any public member could be accessed in any order. Any of the accessible constructors could be used to initialize the object. If a member field might ever be set to `null`, the compiler must assume its *null-state* is *maybe-null* at the start of each method.

You use annotations that can declare whether a variable is a **nullable reference type** or a **non-nullable reference type**. These annotations make important statements about the *null-state* for variables:

- **A reference isn't supposed to be null.** The default state of a nonnullable reference variable is *not-null*. The compiler enforces rules that ensure it's safe to dereference these variables without first checking that it isn't null:
 - The variable must be initialized to a non-null value.
 - The variable can never be assigned the value `null`. The compiler issues a warning when code assigns a *maybe-null* expression to a variable that shouldn't be null.
- **A reference may be null.** The default state of a nullable reference variable is *maybe-null*. The compiler enforces rules to ensure that you've correctly checked for a `null` reference:
 - The variable may only be dereferenced when the compiler can guarantee that the value isn't `null`.
 - These variables may be initialized with the default `null` value and may be assigned the value `null` in other code.
 - The compiler doesn't issue warnings when code assigns a *maybe-null* expression to a variable that may be null.

Any reference variable that isn't supposed to be `null` has a *null-state* of *not-null*. Any reference variable that may be `null` initially has the *null-state* of *maybe-null*.

A **nullable reference type** is noted using the same syntax as [nullable value types](#): a `?` is appended to the type of the variable. For example, the following variable declaration represents a nullable string variable, `name`:

```
string? name;
```

Any variable where the `?` isn't appended to the type name is a **non-nullable reference type**. That includes all reference type variables in existing code when you've enabled this feature. However, any implicitly typed local variables (declared using `var`) are **nullable reference types**. As the preceding sections showed, static analysis determines the *null-state* of local variables to determine if they're *maybe-null*.

Sometimes you must override a warning when you know a variable isn't null, but the compiler determines its *null-state* is *maybe-null*. You use the [null-forgiving operator](#) `!` following a variable name to force the *null-state* to be *not-null*. For example, if you know the `name` variable isn't `null` but the compiler issues a warning, you

can write the following code to override the compiler's analysis:

```
name!.Length;
```

Nullable reference types and nullable value types provide a similar semantic concept: A variable can represent a value or object, or that variable may be `null`. However, nullable reference types and nullable value types are implemented differently: nullable value types are implemented using `System.Nullable<T>`, and nullable reference types are implemented by attributes read by the compiler. For example, `string?` and `string` are both represented by the same type: `System.String`. However, `int?` and `int` are represented by `System.Nullable<System.Int32>` and `System.Int32`, respectively.

Generics

Generics require detailed rules to handle `T?` for any type parameter `T`. The rules are necessarily detailed because of history and the different implementation for a nullable value type and a nullable reference type. [Nullable value types](#) are implemented using the `System.Nullable<T>` struct. [Nullable reference types](#) are implemented as type annotations that provide semantic rules to the compiler.

In C# 8.0, using `T?` without constraining `T` to be a `struct` or a `class` did not compile. That enabled the compiler to interpret `T?` clearly. That restriction was removed in C# 9.0, by defining the following rules for an unconstrained type parameter `T`:

- If the type argument for `T` is a reference type, `T?` references the corresponding nullable reference type. For example, if `T` is a `string`, then `T?` is a `string?`.
- If the type argument for `T` is a value type, `T?` references the same value type, `T`. For example, if `T` is an `int`, the `T?` is also an `int`.
- If the type argument for `T` is a nullable reference type, `T?` references that same nullable reference type. For example, if `T` is a `string?`, then `T?` is also a `string?`.
- If the type argument for `T` is a nullable value type, `T?` references that same nullable value type. For example, if `T` is a `int?`, then `T?` is also a `int?`.

For return values, `T?` is equivalent to `[MaybeNull]T`; for argument values, `T?` is equivalent to `[AllowNull]T`. For more information, see the article on [Attributes for null-state analysis](#) in the language reference.

You can specify different behavior using [constraints](#):

- The `class` constraint means that `T` must be a non-nullable reference type (for example `string`). The compiler produces a warning if you use a nullable reference type, such as `string?` for `T`.
- The `class?` constraint means that `T` must be a reference type, either non-nullable (`string`) or a nullable reference type (for example `string?`). When the type parameter is a nullable reference type, such as `string?`, an expression of `T?` references that same nullable reference type, such as `string?`.
- The `notnull` constraint means that `T` must be a non-nullable reference type, or a non-nullable value type. If you use a nullable reference type or a nullable value type for the type parameter, the compiler produces a warning. Furthermore, when `T` is a value type, the return value is that value type, not the corresponding nullable value type.

These constraints help provide more information to the compiler on how `T` will be used. That helps when developers choose the type for `T`, and provides better *null-state* analysis when an instance of the generic type is used.

Nullable contexts

The new features that protect against throwing a `System.NullReferenceException` can be disruptive when turned

on in an existing codebase:

- All explicitly typed reference variables are interpreted as non-nullable reference types.
- The meaning of the `class` constraint in generics changed to mean a non-nullable reference type.
- New warnings are generated because of these new rules.

You must explicitly opt in to use these features in your existing projects. That provides a migration path and preserves backwards compatibility. Nullable contexts enable fine-grained control for how the compiler interprets reference type variables. The **nullable annotation context** determines the compiler's behavior.

There are four values for the **nullable annotation context**:

- *disabled*: The compiler behaves similarly to C# 7.3 and earlier:
 - Nullable warnings are disabled.
 - All reference type variables are nullable reference types.
 - You can't declare a variable as a nullable reference type using the `?` suffix on the type.
 - You can use the null forgiving operator, `!`, but it has no effect.
- *enabled*: The compiler enables all null reference analysis and all language features.
 - All new nullable warnings are enabled.
 - You can use the `?` suffix to declare a nullable reference type.
 - All other reference type variables are non-nullable reference types.
 - The null forgiving operator suppresses warnings for a possible assignment to `null`.
- *warnings*: The compiler performs all null analysis and emits warnings when code might dereference `null`.
 - All new nullable warnings are enabled.
 - Use of the `?` suffix to declare a nullable reference type produces a warning.
 - All reference type variables are allowed to be null. However, members have the *null-state* of *not-null* at the opening brace of all methods unless declared with the `?` suffix.
 - You can use the null forgiving operator, `!`.
- *annotations*: The compiler doesn't perform null analysis or emit warnings when code might dereference `null`.
 - All new nullable warnings are disabled.
 - You can use the `?` suffix to declare a nullable reference type.
 - All other reference type variables are non-nullable reference types.
 - You can use the null forgiving operator, `!`, but it has no effect.

The nullable annotation context and nullable warning context can be set for a project using the `<Nullable>` element in your `.csproj` file. This element configures how the compiler interprets the nullability of types and what warnings are emitted. The following table shows the allowable values and summarizes the contexts they specify.

CONTEXT	DEREFERENCE WARNINGS	ASSIGNMENT WARNINGS	REFERENCE TYPES	<code>?</code> SUFFIX	<code>!</code> OPERATOR
<code>disabled</code>	Disabled	Disabled	All are nullable	Can't be used	Has no effect
<code>enabled</code>	Enabled	Enabled	Non-nullable unless declared with <code>?</code>	Declares nullable type	Suppresses warnings for possible <code>null</code> assignment

CONTEXT	DEREFERENCE WARNINGS	ASSIGNMENT WARNINGS	REFERENCE TYPES	? SUFFIX	! OPERATOR
warnings	Enabled	Not applicable	All are nullable, but members are considered <i>not null</i> at opening brace of methods	Produces a warning	Suppresses warnings for possible <code>null</code> assignment
annotations	Disabled	Disabled	Non-nullable unless declared with ?	Declares nullable type	Has no effect

Reference type variables in code compiled before C# 8, or in a *disabled* context is *nullable-oblivious*. You can assign a `null` literal or a *maybe-null* variable to a variable that is *nullable oblivious*. However, the default state of a *nullable-oblivious* variable is *not-null*.

You can choose which setting is best for your project:

- Choose *disabled* for legacy projects that you don't want to update based on diagnostics or new features.
- Choose *warnings* to determine where your code may throw [System.NullReferenceExceptions](#). You can address those warnings before modifying code to enable non-nullable reference types.
- Choose *annotations* to express your design intent before enabling warnings.
- Choose *enabled* for new projects and active projects where you want to protect against null reference exceptions.

Example:

```
<Nullable>enable</Nullable>
```

You can also use directives to set these same contexts anywhere in your source code. These are most useful when you're migrating a large codebase.

- `#nullable enable`: Sets the nullable annotation context and nullable warning context to **enabled**.
- `#nullable disable`: Sets the nullable annotation context and nullable warning context to **disabled**.
- `#nullable restore`: Restores the nullable annotation context and nullable warning context to the project settings.
- `#nullable disable warnings`: Set the nullable warning context to **disabled**.
- `#nullable enable warnings`: Set the nullable warning context to **enabled**.
- `#nullable restore warnings`: Restores the nullable warning context to the project settings.
- `#nullable disable annotations`: Set the nullable annotation context to **disabled**.
- `#nullable enable annotations`: Set the nullable annotation context to **enabled**.
- `#nullable restore annotations`: Restores the annotation warning context to the project settings.

For any line of code, you can set any of the following combinations:

WARNING CONTEXT	ANNOTATION CONTEXT	USE
project default	project default	Default
enabled	disabled	Fix analysis warnings

WARNING CONTEXT	ANNOTATION CONTEXT	USE
enabled	project default	Fix analysis warnings
project default	enabled	Add type annotations
enabled	enabled	Code already migrated
disabled	enabled	Annotate code before fixing warnings
disabled	disabled	Adding legacy code to migrated project
project default	disabled	Rarely
disabled	project default	Rarely

Those nine combinations provide you with fine-grained control over the diagnostics the compiler emits for your code. You can enable more features in any area you're updating, without seeing additional warnings you aren't ready to address yet.

IMPORTANT

The global nullable context does not apply for generated code files. Under either strategy, the nullable context is *disabled* for any source file marked as generated. This means any APIs in generated files are not annotated. There are four ways a file is marked as generated:

1. In the .editorconfig, specify `generated_code = true` in a section that applies to that file.
2. Put `<auto-generated>` or `<auto-generated/>` in a comment at the top of the file. It can be on any line in that comment, but the comment block must be the first element in the file.
3. Start the file name with *TemporaryGeneratedFile_*
4. End the file name with *.designer.cs*, *.generated.cs*, *.g.cs*, or *.g.i.cs*.

Generators can opt-in using the `#nullable` preprocessor directive.

By default, nullable annotation and warning contexts are **disabled**. That means that your existing code compiles without changes and without generating any new warnings. Beginning with .NET 6, new projects include the `<Nullable>enable</Nullable>` element in all project templates.

These options provide two distinct strategies to [update an existing codebase](#) to use nullable reference types.

Known pitfalls

Arrays and structs that contain reference types are known pitfalls in nullable references and the static analysis that determines null safety. In both situations, a non-nullable reference may be initialized to `null`, without generating warnings.

Structs

A struct that contains non-nullable reference types allows assigning `default` for it without any warnings. Consider the following example:

```

using System;

#nullable enable

public struct Student
{
    public string FirstName;
    public string? MiddleName;
    public string LastName;
}

public static class Program
{
    public static void PrintStudent(Student student)
    {
        Console.WriteLine($"First name: {student.FirstName.ToUpper()}");
        Console.WriteLine($"Middle name: {student.MiddleName?.ToUpper()}");
        Console.WriteLine($"Last name: {student.LastName.ToUpper()}");
    }

    public static void Main() => PrintStudent(default);
}

```

In the preceding example, there's no warning in `PrintStudent(default)` while the non-nullable reference types `FirstName` and `LastName` are null.

Another more common case is when you deal with generic structs. Consider the following example:

```

#nullable enable

public struct Foo<T>
{
    public T Bar { get; set; }
}

public static class Program
{
    public static void Main()
    {
        string s = default(Foo<string>).Bar;
    }
}

```

In the preceding example, the property `Bar` is going to be `null` at run time, and it's assigned to non-nullable string without any warnings.

Arrays

Arrays are also a known pitfall in nullable reference types. Consider the following example that doesn't produce any warnings:

```
using System;

#nullable enable

public static class Program
{
    public static void Main()
    {
        string[] values = new string[10];
        string s = values[0];
        Console.WriteLine(s.ToUpper());
    }
}
```

In the preceding example, the declaration of the array shows it holds non-nullable strings, while its elements are all initialized to `null`. Then, the variable `s` is assigned a `null` value (the first element of the array). Finally, the variable `s` is dereferenced causing a runtime exception.

See also

- [Nullable reference types proposal](#)
- [Draft nullable reference types specification](#)
- [Unconstrained type parameter annotations](#)
- [Intro to nullable references tutorial](#)
- [Nullable \(C# Compiler option\)](#)

Update a codebase with nullable reference types to improve null diagnostic warnings

12/28/2021 • 6 minutes to read • [Edit Online](#)

[Nullable reference types](#) enable you to declare if variables of a reference type should or shouldn't be assigned a `null` value. The compiler's static analysis and warnings when your code might dereference `null` are the most important benefit of this feature. Once enabled, the compiler generates warnings that help you avoid throwing a [System.NullReferenceException](#) when your code runs.

If your codebase is relatively small, you can turn on the [feature in your project](#), address warnings, and enjoy the benefits of the improved diagnostics. Larger codebases may require a more structured approach to address warnings over time, enabling the feature for some as you address warnings in different types or files. This article describes different strategies to update a codebase and the tradeoffs associated with these strategies. Before starting your migration, read the conceptual overview of [nullable reference types](#). It covers the compiler's static analysis, *null-state* values of *maybe-null* and *not-null* and the nullable annotations. Once you're familiar with those concepts and terms, you're ready to migrate your code.

Plan your migration

Regardless of how you update your codebase, the goal is that nullable warnings and nullable annotations are enabled in your project. Once you reach that goal, you'll have the `<nullable>Enable</nullable>` setting in your project. You won't need any of the pragmas to adjust settings elsewhere.

The first choice is setting the default for the project. Your choices are:

1. **Nullable disable as the default:** *disable* is the default if you don't add a `Nullable` element to your project file. Use this default when you're not actively adding new files to the codebase. The main activity is to update the library to use nullable reference types. Using this default means you add a nullable pragma to each file as you update its code.
2. **Nullable enable as the default:** Set this default when you're actively developing new features. You want all new code to benefit nullable reference types and nullable static analysis. Using this default means you must add a `#pragma nullable disable` to the top of each file. You'll remove that pragma as you begin addressing the warnings in that file.
3. **Nullable warnings as the default:** Choose this default for a two-phase migration. In the first phase, address warnings. In the second phase, turn on annotations for declaring a variable's expected *null-state*. Using this default means you must add a `#pragma nullable disable` to the top of each file.
4. **Nullable annotations as the default:** Annotate code before addressing warnings.

Enabling nullable as the default creates more up-front work to add the pragma to every file. The advantage is that every new code file added to the project will be nullable enabled. Any new work will be nullable aware; only existing code must be updated. Disabling nullable as the default works better if the library is stable, and the main focus of the development is to adopt nullable reference types. You turn on nullable reference types as you annotate APIs. When you've finished, you enable nullable reference types for the entire project. When you create a new file, you must add the pragmas and make it nullable aware. If any developers on your team forget, that new code is now in the backlog of work to make all code nullable aware.

Which of these strategies you pick depends on how much active development is taking place in your project. The more mature and stable your project, the better the second strategy. The more features being developed, the better the first strategy.

IMPORTANT

The global nullable context does not apply for generated code files. Under either strategy, the nullable context is *disabled* for any source file marked as generated. This means any APIs in generated files are not annotated. There are four ways a file is marked as generated:

1. In the .editorconfig, specify `generated_code = true` in a section that applies to that file.
2. Put `<auto-generated>` or `<auto-generated/>` in a comment at the top of the file. It can be on any line in that comment, but the comment block must be the first element in the file.
3. Start the file name with `TemporaryGeneratedFile_`
4. End the file name with `.designer.cs`, `.generated.cs`, `.g.cs`, or `.g.i.cs`.

Generators can opt-in using the `#nullable` preprocessor directive.

Understand contexts and warnings

Enabling warnings and annotations control how the compiler views reference types and nullability. Every type has one of three nullabilities:

- *oblivious*: All reference types are nullable *oblivious* when the annotation context is disabled.
- *nonnullable*: An unannotated reference type, `C`, is *nonnullable* when the annotation context is enabled.
- *nullable*: An annotated reference type, `C?`, is *nullable*, but a warning may be issued when the annotation context is disabled. Variables declared with `var` are *nullable* when the annotation context is enabled.

The compiler generates warnings based on that nullability:

- *nonnullable* types cause warnings if a potential `null` value is assigned to them.
- *nullable* types cause warnings if they are dereferenced when *maybe-null*.
- *oblivious* types cause warnings if they're dereferenced when *maybe-null* and the warning context is enabled.

Each variable has a default nullable state that depends on its nullability:

- Nullable variables have a default *null-state* of *maybe-null*.
- Non-nullable variables have a default *null-state* of *not-null*.
- Nullable oblivious variables have a default *null-state* of *not-null*.

Before you enable nullable reference types, all declarations in your codebase are *nullable oblivious*. That's important because it means all reference types have a default *null-state* of *not-null*.

Address warnings

If your project uses Entity Framework Core, you should read their guidance on [Working with nullable reference types](#).

When you start your migration, you should start by enabling warnings only. All declarations remain *nullable oblivious*, but you'll see warnings when you dereference a value after its *null-state* changes to *maybe-null*. As you address these warnings, you'll be checking against null in more locations, and your codebase becomes more resilient. To learn specific techniques for different situations, see the article on [Techniques to resolve nullable warnings](#).

You can address warnings and enable annotations in each file or class before continuing with other code. However, it's often more efficient to address the warnings generated while the context is *warnings* before enabling the type annotations. That way, all types are *oblivious* until you've addressed the first set of warnings.

Enable type annotations

After addressing the first set of warnings, you can enable the *annotation context*. This changes reference types from *oblivious* to *nonnullable*. All variables declared with `var` are *nullable*. This change often introduces new warnings. The first step in addressing the compiler warnings is to use `?` annotations on parameter and return types to indicate when arguments or return values may be `null`. As you do this task, your goal isn't just to fix warnings. The more important goal is to make the compiler understand your intent for potential null values.

Attributes extend type annotations

Several attributes have been added to express additional information about the null state of variables. The rules for your APIs are likely more complicated than *not-null* or *maybe-null* for all parameters and return values. Many of your APIs have more complex rules for when variables can or can't be `null`. In these cases, you'll use attributes to express those rules. The attributes that describe the semantics of your API are found in the article on [Attributes that affect nullable analysis](#).

Next steps

Once you've addressed all warnings after enabling annotations, you can set the default context for your project to *enabled*. If you added any pragmas in your code for the nullable annotation or warning context, you can remove them. Over time, you may see new warnings. You may write code that introduces warnings. A library dependency may be updated for nullable reference types. Those updates will change the types in that library from *nullable oblivious* to either *nonnullable* or *nullable*.

Learn techniques to resolve nullable warnings

12/28/2021 • 7 minutes to read • [Edit Online](#)

The purpose of nullable reference types is to minimize the chance that your application throws a [System.NullReferenceException](#) when run. To achieve this goal, the compiler uses static analysis and issues warnings when your code has constructs that may lead to null reference exceptions. You provide the compiler with information for its static analysis by applying type annotations and attributes. These annotations and attributes describe the nullability of arguments, parameters, and members of your types. In this article, you'll learn different techniques to address the nullable warnings the compiler generates from its static analysis. The techniques described here are for general C# code. Learn to work with nullable reference types and Entity Framework core in [Working with nullable reference types](#).

You'll address almost all warnings using one of four techniques:

- Adding necessary null checks.
- Adding `?` or `!` nullable annotations.
- Adding attributes that describe null semantics.
- Initializing variables correctly.

Possible dereference of null

One set of warnings alert you that you're dereferencing a variable whose *null-state* is *maybe-null*. One example might be:

```
string message = null;
Console.WriteLine(message.Length);
```

To remove these warnings, you need to add code to change that variable's *null-state* to *not-null* before dereferencing it.

In many instances, you can fix these warnings by checking that a variable isn't null before dereferencing it. For example, the above example could be rewritten as:

```
string message = null;
if (message is not null)
{
    Console.WriteLine(message.Length);
}
```

When your code generates a warning that it may be dereferencing a *maybe-null* reference, make sure you've done a null check. If you haven't, add one. The compiler warning helped you address a possible bug.

Other instances when you get these warnings may be false positive. You may have a private utility method that tests for null. The compiler doesn't know that the method provides a null check. Consider the following example that uses a private utility method, `IsNotNull`:


```
public void WriteMessage(string? message)
{
    if (IsNotNull(message))
        Console.WriteLine(message.Length);
}
```

The compiler warns that you may be dereferencing null when you write the property `message.Length` because its static analysis determines that `message` may be `null`. You may know that `IsNotNull` provides a null check, and when it returns `true`, the *null-state* of `message` should be *not-null*. You must tell the compiler those facts. One way is to use the null forgiving operator, `!`. You can change the `WriteLine` statement to match the following code:

```
Console.WriteLine(message!.Length);
```

The null forgiving operator makes the expression *not-null* even if it was *maybe-null* without the `!` applied. In this example, a better solution is to add an attribute to the signature of `IsNotNull`:

```
private static bool IsNotNull([NotNullWhen(true)] object? obj) => obj != null;
```

The [System.Diagnostics.CodeAnalysis.NotNullWhenAttribute](#) informs the compiler that the argument used for the `obj` parameter is *not-null* when the method returns `true`. When the method returns `false`, the argument has the same *null-state* it had before the method was called.

TIP

There's a rich set of attributes you can use to describe how your methods and properties affect *null-state*. You can learn about them in the language reference article on [Nullable static analysis attributes](#).

Fixing a warning for dereferencing a *maybe-null* variable involves one of three techniques:

- Add a missing null check.
- Add null analysis attributes on APIs to affect the compiler's *null-state* static analysis. These attributes inform the compiler when a return value or argument should be *maybe-null* or *not-null* after calling the method.
- Apply the null forgiving operator `!` to the expression to force the state to *not-null*.

Possible null assigned to a nonnullable reference

The compiler emits these warnings when you attempt to assign an expression that is *maybe-null* to a variable that is nonnullable. For example:

```
string? TryGetMessage(int id) => "";

string msg = TryGetMessage(42); // Possible null assignment.
```

You can take one of three actions to address these warnings. One is to add the `?` annotation to make the variable a nullable reference type. That change may cause other warnings. Changing a variable from a non-nullable reference to a nullable reference changes its default *null-state* from *not-null* to *maybe-null*. The compiler's static analysis may find instances where you dereference a variable that is *maybe-null*.

The other actions instruct the compiler that the right-hand-side of the assignment is *not-null*. The expression on the right-hand-side could be null checked before assignment, as shown in the following example:

```
string notNullMsg = TryGetMessage(42) ?? "Unknown message id: 42";
```

The previous examples demonstrate assignment to the return value of a method. You may annotate the method (or property) to indicate when a method returns a not-null value. The [System.Diagnostics.CodeAnalysis.NotNullIfNotNullAttribute](#) often specifies that a return value is *not-null* when an input argument is *not-null*. Another alternative is to add the null forgiving operator, `!` to the right-hand side:

```
string msg = TryGetMessage(42)!;
```

Fixing a warning for assigning a *maybe-null* expression to a *not-null* variable involves one of four techniques:

- Change the left side of the assignment to a nullable type. This action may introduce new warnings when you dereference that variable.
- Provide a null-check before the assignment.
- Annotate the API that produces the right-hand side of the assignment.
- Add the null forgiving operator to the right-hand side of the assignment.

Nonnullable reference not initialized

Other warnings are generated when a nonnullable reference variable isn't initialized when declared, or in a constructor. Consider the following class as an example:

```
public class Person
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
}
```

Neither `FirstName` nor `LastName` are guaranteed initialized. If this code is new, consider changing the public interface. The above example could be updated as follows:

```
public class Person
{
    public Person(string first, string last)
    {
        FirstName = first;
        LastName = last;
    }

    public string FirstName { get; set; }
    public string LastName { get; set; }
}
```

If you require creating a `Person` object before setting the name, you can initialize the properties using a default non-null value:

```
public class Person
{
    public string FirstName { get; set; } = string.Empty;
    public string LastName { get; set; } = string.Empty;
}
```

Another alternative may be to change those members to nullable reference types. The `Person` class could be defined as follows if `null` should be allowed for the name:

```
public class Person
{
    public string? FirstName { get; set; }
    public string? LastName { get; set; }
}
```

Existing code may require other changes to inform the compiler about the null semantics for those members. You may have created multiple constructors, and your class may have a private helper method that initializes one or more members. The [System.Diagnostics.CodeAnalysis.MemberNotNullAttribute](#) and [System.Diagnostics.CodeAnalysis.MemberNotNullWhenAttribute](#) attributes inform the compiler that a member is *not-null* after the method has been called.

Finally, you can use the null forgiving operator to indicate that a member is initialized in other code. For another example, consider the following classes representing an Entity Framework Core model:

```
public class TodoItem
{
    public long Id { get; set; }
    public string? Name { get; set; }
    public bool IsComplete { get; set; }
}

public class TodoContext : DbContext
{
    public TodoContext(DbContextOptions<TodoContext> options)
        : base(options)
    {
    }

    public DbSet<TodoItem> TodoItems { get; set; } = null!;
}
```

The `DbSet` property is initialized to `null!`. That tells the compiler that the property is set to a *not-null* value. In fact, the base `DbContext` performs the initialization of the set. The compiler's static analysis doesn't pick that up. For more information on working with nullable reference types and Entity Framework Core, see the article on [Working with Nullable Reference Types in EF Core](#).

Fixing a warning for not initializing a nonnullable member involves one of four techniques:

- Change the constructors or field initializers to ensure all nonnullable members are initialized.
- Change one or more members to be nullable types.
- Annotate any helper methods to indicate which members are assigned.
- Add an initializer to `null!` to indicate that the member is initialized in other code.

Mismatch in nullability declaration

Other warnings indicate nullability mismatches between signatures for methods, delegates, or type parameters. For example:

```
public class B
{
    public virtual string GetMessage(string id) => string.Empty;
}

public class D : B
{
    public override string? GetMessage(string? id) => default;
}
```

The preceding example shows a `virtual` method in a base class and an `override` with different nullability. The base class returns a non-nullable string, but the derived class returns a nullable string. If the `string` and `string?` are reversed, it would be allowed because the derived class is more restrictive. Similarly, parameter declarations should match. Parameters in the override method can allow null even when the base class doesn't.

Other situations can generate these warnings. You may have a mismatch in an interface method declaration and the implementation of that method. Or a delegate type and the expression for that delegate may differ. A type parameter and the type argument may differ in nullability.

To fix these warnings, update the appropriate declaration.

Code doesn't match attribute declaration

The preceding sections have discussed how you can use [Attributes for nullable static analysis](#) to inform the compiler about the null semantics of your code. The compiler warns you if the code doesn't adhere to the promises of that attribute. Consider the following method:

```
public bool TryGetMessage(int id, [NotNullWhen(true)] out string? message)
{
    message = null;
    return true;
}
```

The compiler produces a warning because the `message` parameter is assigned `null` and the method returns `true`. The `NotNullWhen` attribute indicates that shouldn't happen.

To address these warnings, update your code so it matches the expectations of the attributes you've applied. You may change the attributes, or the algorithm.

Methods in (C#)

12/28/2021 • 22 minutes to read • [Edit Online](#)

A method is a code block that contains a series of statements. A program causes the statements to be executed by calling the method and specifying any required method arguments. In C#, every executed instruction is performed in the context of a method. The `Main` method is the entry point for every C# application and it is called by the common language runtime (CLR) when the program is started.

NOTE

This topic discusses named methods. For information about anonymous functions, see [Lambda expressions](#).

Method signatures

Methods are declared in a `class`, `record`, or `struct` by specifying:

- An optional access level, such as `public` or `private`. The default is `private`.
- Optional modifiers such as `abstract` or `sealed`.
- The return value, or `void` if the method has none.
- The method name.
- Any method parameters. Method parameters are enclosed in parentheses and are separated by commas. Empty parentheses indicate that the method requires no parameters.

These parts together form the method signature.

IMPORTANT

A return type of a method is not part of the signature of the method for the purposes of method overloading. However, it is part of the signature of the method when determining the compatibility between a delegate and the method that it points to.

The following example defines a class named `Motorcycle` that contains five methods:

```

using System;

abstract class Motorcycle
{
    // Anyone can call this.
    public void StartEngine() { /* Method statements here */ }

    // Only derived classes can call this.
    protected void AddGas(int gallons) { /* Method statements here */ }

    // Derived classes can override the base class implementation.
    public virtual int Drive(int miles, int speed) { /* Method statements here */ return 1; }

    // Derived classes can override the base class implementation.
    public virtual int Drive(TimeSpan time, int speed) { /* Method statements here */ return 0; }

    // Derived classes must implement this.
    public abstract double GetTopSpeed();
}

```

Note that the `Motorcycle` class includes an overloaded method, `Drive`. Two methods have the same name, but must be differentiated by their parameter types.

Method invocation

Methods can be either *instance* or *static*. Invoking an instance method requires that you instantiate an object and call the method on that object; an instance method operates on that instance and its data. You invoke a static method by referencing the name of the type to which the method belongs; static methods do not operate on instance data. Attempting to call a static method through an object instance generates a compiler error.

Calling a method is like accessing a field. After the object name (if you are calling an instance method) or the type name (if you are calling a `static` method), add a period, the name of the method, and parentheses. Arguments are listed within the parentheses and are separated by commas.

The method definition specifies the names and types of any parameters that are required. When a caller invokes the method, it provides concrete values, called arguments, for each parameter. The arguments must be compatible with the parameter type, but the argument name, if one is used in the calling code, does not have to be the same as the parameter named defined in the method. In the following example, the `Square` method includes a single parameter of type `int` named *i*. The first method call passes the `Square` method a variable of type `int` named *num*; the second, a numeric constant; and the third, an expression.

```

public class SquareExample
{
    public static void Main()
    {
        // Call with an int variable.
        int num = 4;
        int productA = Square(num);

        // Call with an integer literal.
        int productB = Square(12);

        // Call with an expression that evaluates to int.
        int productC = Square(productA * 3);
    }

    static int Square(int i)
    {
        // Store input argument in a local variable.
        int input = i;
        return input * input;
    }
}

```

The most common form of method invocation used positional arguments; it supplies arguments in the same order as method parameters. The methods of the `Motorcycle` class can therefore be called as in the following example. The call to the `Drive` method, for example, includes two arguments that correspond to the two parameters in the method's syntax. The first becomes the value of the `miles` parameter, the second the value of the `speed` parameter.

```

class TestMotorcycle : Motorcycle
{
    public override double GetTopSpeed()
    {
        return 108.4;
    }

    static void Main()
    {
        TestMotorcycle moto = new TestMotorcycle();

        moto.StartEngine();
        moto.AddGas(15);
        moto.Drive(5, 20);
        double speed = moto.GetTopSpeed();
        Console.WriteLine("My top speed is {0}", speed);
    }
}

```

You can also use *named arguments* instead of positional arguments when invoking a method. When using named arguments, you specify the parameter name followed by a colon (":") and the argument. Arguments to the method can appear in any order, as long as all required arguments are present. The following example uses named arguments to invoke the `TestMotorcycle.Drive` method. In this example, the named arguments are passed in the opposite order from the method's parameter list.

```

using System;

class TestMotorcycle : Motorcycle
{
    public override int Drive(int miles, int speed)
    {
        return (int)Math.Round(((double)miles) / speed, 0);
    }

    public override double GetTopSpeed()
    {
        return 108.4;
    }

    static void Main()
    {
        TestMotorcycle moto = new TestMotorcycle();
        moto.StartEngine();
        moto.AddGas(15);
        var travelTime = moto.Drive(speed: 60, miles: 170);
        Console.WriteLine("Travel time: approx. {0} hours", travelTime);
    }
}
// The example displays the following output:
//      Travel time: approx. 3 hours

```

You can invoke a method using both positional arguments and named arguments. However, positional arguments can only follow named arguments when the named arguments are in the correct positions. The following example invokes the `TestMotorcycle.Drive` method from the previous example using one positional argument and one named argument.

```

var travelTime = moto.Drive(170, speed: 55);

```

Inherited and overridden methods

In addition to the members that are explicitly defined in a type, a type inherits members defined in its base classes. Since all types in the managed type system inherit directly or indirectly from the `Object` class, all types inherit its members, such as `Equals(Object)`, `GetType()`, and `ToString()`. The following example defines a `Person` class, instantiates two `Person` objects, and calls the `Person.Equals` method to determine whether the two objects are equal. The `Equals` method, however, is not defined in the `Person` class; it is inherited from `Object`.


```

using System;

public class Person
{
    public String FirstName;
}

public class ClassTypeExample
{
    public static void Main()
    {
        var p1 = new Person();
        p1.FirstName = "John";
        var p2 = new Person();
        p2.FirstName = "John";
        Console.WriteLine("p1 = p2: {0}", p1.Equals(p2));
    }
}

// The example displays the following output:
//      p1 = p2: False

```

Types can override inherited members by using the `override` keyword and providing an implementation for the overridden method. The method signature must be the same as that of the overridden method. The following example is like the previous one, except that it overrides the [Equals\(Object\)](#) method. (It also overrides the [GetHashCode\(\)](#) method, since the two methods are intended to provide consistent results.)

```

using System;

public class Person
{
    public String FirstName;

    public override bool Equals(object obj)
    {
        var p2 = obj as Person;
        if (p2 == null)
            return false;
        else
            return FirstName.Equals(p2.FirstName);
    }

    public override int GetHashCode()
    {
        return FirstName.GetHashCode();
    }
}

public class Example
{
    public static void Main()
    {
        var p1 = new Person();
        p1.FirstName = "John";
        var p2 = new Person();
        p2.FirstName = "John";
        Console.WriteLine("p1 = p2: {0}", p1.Equals(p2));
    }
}

// The example displays the following output:
//      p1 = p2: True

```

Passing parameters

Types in C# are either *value types* or *reference types*. For a list of built-in value types, see [Types](#). By default, both value types and reference types are passed to a method by value.

Passing parameters by value

When a value type is passed to a method by value, a copy of the object instead of the object itself is passed to the method. Therefore, changes to the object in the called method have no effect on the original object when control returns to the caller.

The following example passes a value type to a method by value, and the called method attempts to change the value type's value. It defines a variable of type `int`, which is a value type, initializes its value to 20, and passes it to a method named `ModifyValue` that changes the variable's value to 30. When the method returns, however, the variable's value remains unchanged.

```
using System;

public class ByValueExample
{
    public static void Main()
    {
        int value = 20;
        Console.WriteLine("In Main, value = {0}", value);
        ModifyValue(value);
        Console.WriteLine("Back in Main, value = {0}", value);
    }

    static void ModifyValue(int i)
    {
        i = 30;
        Console.WriteLine("In ModifyValue, parameter value = {0}", i);
        return;
    }
}

// The example displays the following output:
//      In Main, value = 20
//      In ModifyValue, parameter value = 30
//      Back in Main, value = 20
```

When an object of a reference type is passed to a method by value, a reference to the object is passed by value. That is, the method receives not the object itself, but an argument that indicates the location of the object. If you change a member of the object by using this reference, the change is reflected in the object when control returns to the calling method. However, replacing the object passed to the method has no effect on the original object when control returns to the caller.

The following example defines a class (which is a reference type) named `SampleRefType`. It instantiates a `SampleRefType` object, assigns 44 to its `value` field, and passes the object to the `ModifyObject` method. This example does essentially the same thing as the previous example -- it passes an argument by value to a method. But because a reference type is used, the result is different. The modification that is made in `ModifyObject` to the `obj.value` field also changes the `value` field of the argument, `rt`, in the `Main` method to 33, as the output from the example shows.

```

using System;

public class SampleRefType
{
    public int value;
}

public class ByRefTypeExample
{
    public static void Main()
    {
        var rt = new SampleRefType();
        rt.value = 44;
        ModifyObject(rt);
        Console.WriteLine(rt.value);
    }

    static void ModifyObject(SampleRefType obj)
    {
        obj.value = 33;
    }
}

```

Passing parameters by reference

You pass a parameter by reference when you want to change the value of an argument in a method and want to reflect that change when control returns to the calling method. To pass a parameter by reference, you use the `ref` or `out` keyword. You can also pass a value by reference to avoid copying but still prevent modifications using the `in` keyword.

The following example is identical to the previous one, except the value is passed by reference to the `ModifyValue` method. When the value of the parameter is modified in the `ModifyValue` method, the change in value is reflected when control returns to the caller.

```

using System;

public class ByRefExample
{
    public static void Main()
    {
        int value = 20;
        Console.WriteLine("In Main, value = {0}", value);
        ModifyValue(ref value);
        Console.WriteLine("Back in Main, value = {0}", value);
    }

    static void ModifyValue(ref int i)
    {
        i = 30;
        Console.WriteLine("In ModifyValue, parameter value = {0}", i);
        return;
    }
}

// The example displays the following output:
//      In Main, value = 20
//      In ModifyValue, parameter value = 30
//      Back in Main, value = 30

```

A common pattern that uses by ref parameters involves swapping the values of variables. You pass two variables to a method by reference, and the method swaps their contents. The following example swaps integer values.

```

using System;

public class RefSwapExample
{
    static void Main()
    {
        int i = 2, j = 3;
        System.Console.WriteLine("i = {0} j = {1}" , i, j);

        Swap(ref i, ref j);

        System.Console.WriteLine("i = {0} j = {1}" , i, j);
    }

    static void Swap(ref int x, ref int y)
    {
        int temp = x;
        x = y;
        y = temp;
    }
}

// The example displays the following output:
//      i = 2 j = 3
//      i = 3 j = 2

```

Passing a reference-type parameter allows you to change the value of the reference itself, rather than the value of its individual elements or fields.

Parameter arrays

Sometimes, the requirement that you specify the exact number of arguments to your method is restrictive. By using the `params` keyword to indicate that a parameter is a parameter array, you allow your method to be called with a variable number of arguments. The parameter tagged with the `params` keyword must be an array type, and it must be the last parameter in the method's parameter list.

A caller can then invoke the method in either of four ways:

- By passing an array of the appropriate type that contains the desired number of elements.
- By passing a comma-separated list of individual arguments of the appropriate type to the method.
- By passing `null`.
- By not providing an argument to the parameter array.

The following example defines a method named `GetVowels` that returns all the vowels from a parameter array. The `Main` method illustrates all four ways of invoking the method. Callers are not required to supply any arguments for parameters that include the `params` modifier. In that case, the parameter is an empty array.

```

using System;
using System.Linq;

class ParamsExample
{
    static void Main()
    {
        string fromArray = GetVowels(new[] { "apple", "banana", "pear" });
        Console.WriteLine($"Vowels from array: '{fromArray}'");

        string fromMultipleArguments = GetVowels("apple", "banana", "pear");
        Console.WriteLine($"Vowels from multiple arguments: '{fromMultipleArguments}'");

        string fromNull = GetVowels(null);
        Console.WriteLine($"Vowels from null: '{fromNull}'");

        string fromNoValue = GetVowels();
        Console.WriteLine($"Vowels from no value: '{fromNoValue}'");
    }

    static string GetVowels(params string[] input)
    {
        if (input == null || input.Length == 0)
        {
            return string.Empty;
        }

        var vowels = new char[] { 'A', 'E', 'I', 'O', 'U' };
        return string.Concat(
            input.SelectMany(
                word => word.Where(letter => vowels.Contains(char.ToUpper(letter)))));
    }
}

// The example displays the following output:
//     Vowels from array: 'aeaaaaa'
//     Vowels from multiple arguments: 'aeaaaaa'
//     Vowels from null: ''
//     Vowels from no value: ''

```

Optional parameters and arguments

A method definition can specify that its parameters are required or that they are optional. By default, parameters are required. Optional parameters are specified by including the parameter's default value in the method definition. When the method is called, if no argument is supplied for an optional parameter, the default value is used instead.

The parameter's default value must be assigned by one of the following kinds of expressions:

- A constant, such as a literal string or number.
- An expression of the form `default(SomeType)`, where `SomeType` can be either a value type or a reference type. If it's a reference type, it's effectively the same as specifying `null`. Beginning with C# 7.1, you can use the `default` literal, as the compiler can infer the type from the parameter's declaration.
- An expression of the form `new ValType()`, where `ValType` is a value type. Note that this invokes the value type's implicit parameterless constructor, which is not an actual member of the type.

NOTE

In C# 10 and later, when an expression of the form `new ValType()` invokes the explicitly defined parameterless constructor of a value type, the compiler generates an error as the default parameter value must be a compile-time constant. Use the `default(ValType)` expression or the `default` literal to provide the default parameter value. For more information about parameterless constructors, see the [Parameterless constructors and field initializers](#) section of the [Structure types](#) article.

If a method includes both required and optional parameters, optional parameters are defined at the end of the parameter list, after all required parameters.

The following example defines a method, `ExampleMethod`, that has one required and two optional parameters.

```
using System;

public class Options
{
    public void ExampleMethod(int required, int optionalInt = default,
                             string? description = default)
    {
        var msg = $"{description ?? "N/A"}: {required} + {optionalInt} = {required + optionalInt}";
        Console.WriteLine(msg);
    }
}
```

If a method with multiple optional arguments is invoked using positional arguments, the caller must supply an argument for all optional parameters from the first one to the last one for which an argument is supplied. In the case of the `ExampleMethod` method, for example, if the caller supplies an argument for the `description` parameter, it must also supply one for the `optionalInt` parameter.

`opt.ExampleMethod(2, 2, "Addition of 2 and 2");` is a valid method call;

`opt.ExampleMethod(2, , "Addition of 2 and 0");` generates an "Argument missing" compiler error.

If a method is called using named arguments or a combination of positional and named arguments, the caller can omit any arguments that follow the last positional argument in the method call.

The following example calls the `ExampleMethod` method three times. The first two method calls use positional arguments. The first omits both optional arguments, while the second omits the last argument. The third method call supplies a positional argument for the required parameter but uses a named argument to supply a value to the `description` parameter while omitting the `optionalInt` argument.

```
public class OptionsExample
{
    public static void Main()
    {
        var opt = new Options();
        opt.ExampleMethod(10);
        opt.ExampleMethod(10, 2);
        opt.ExampleMethod(12, description: "Addition with zero:");
    }
}

// The example displays the following output:
//      N/A: 10 + 0 = 10
//      N/A: 10 + 2 = 12
//      Addition with zero:: 12 + 0 = 12
```

The use of optional parameters affects *overload resolution*, or the way in which the C# compiler determines which particular overload should be invoked by a method call, as follows:

- A method, indexer, or constructor is a candidate for execution if each of its parameters either is optional or corresponds, by name or by position, to a single argument in the calling statement, and that argument can be converted to the type of the parameter.
- If more than one candidate is found, overload resolution rules for preferred conversions are applied to the arguments that are explicitly specified. Omitted arguments for optional parameters are ignored.
- If two candidates are judged to be equally good, preference goes to a candidate that does not have optional parameters for which arguments were omitted in the call. This is a consequence of a general preference in overload resolution for candidates that have fewer parameters.

Return values

Methods can return a value to the caller. If the return type (the type listed before the method name) is not `void`, the method can return the value by using the `return` keyword. A statement with the `return` keyword followed by a variable, constant, or expression that matches the return type will return that value to the method caller. Methods with a non-void return type are required to use the `return` keyword to return a value. The `return` keyword also stops the execution of the method.

If the return type is `void`, a `return` statement without a value is still useful to stop the execution of the method. Without the `return` keyword, the method will stop executing when it reaches the end of the code block.

For example, these two methods use the `return` keyword to return integers:

```
class SimpleMath
{
    public int AddTwoNumbers(int number1, int number2)
    {
        return number1 + number2;
    }

    public int SquareANumber(int number)
    {
        return number * number;
    }
}
```

To use a value returned from a method, the calling method can use the method call itself anywhere a value of the same type would be sufficient. You can also assign the return value to a variable. For example, the following two code examples accomplish the same goal:

```
int result = obj.AddTwoNumbers(1, 2);
result = obj.SquareANumber(result);
// The result is 9.
Console.WriteLine(result);
```

```
result = obj.SquareANumber(obj.AddTwoNumbers(1, 2));
// The result is 9.
Console.WriteLine(result);
```

Using a local variable, in this case, `result`, to store a value is optional. It may help the readability of the code, or it may be necessary if you need to store the original value of the argument for the entire scope of the method.

Sometimes, you want your method to return more than a single value. Starting with C# 7.0, you can do this easily by using *tuple types* and *tuple literals*. The tuple type defines the data types of the tuple's elements. Tuple literals provide the actual values of the returned tuple. In the following example, `(string, string, string, int)`

defines the tuple type that is returned by the `GetPersonalInfo` method. The expression `(per.FirstName, per.MiddleName, per.LastName, per.Age)` is the tuple literal; the method returns the first, middle, and last name, along with the age, of a `PersonInfo` object.

```
public (string, string, string, int) GetPersonalInfo(string id)
{
    PersonInfo per = PersonInfo.RetrieveInfoById(id);
    return (per.FirstName, per.MiddleName, per.LastName, per.Age);
}
```

The caller can then consume the returned tuple with code like the following:

```
var person = GetPersonalInfo("111111111");
Console.WriteLine($"{person.Item1} {person.Item3}: age = {person.Item4}");
```

Names can also be assigned to the tuple elements in the tuple type definition. The following example shows an alternate version of the `GetPersonalInfo` method that uses named elements:

```
public (string FName, string MName, string LName, int Age) GetPersonalInfo(string id)
{
    PersonInfo per = PersonInfo.RetrieveInfoById(id);
    return (per.FirstName, per.MiddleName, per.LastName, per.Age);
}
```

The previous call to the `GetPersonalInfo` method can then be modified as follows:

```
var person = GetPersonalInfo("111111111");
Console.WriteLine($"{person.FName} {person.LName}: age = {person.Age}");
```

If a method is passed an array as an argument and modifies the value of individual elements, it is not necessary for the method to return the array, although you may choose to do so for good style or functional flow of values. This is because C# passes all reference types by value, and the value of an array reference is the pointer to the array. In the following example, changes to the contents of the `values` array that are made in the `DoubleValues` method are observable by any code that has a reference to the array.

```
using System;

public class ArrayValueExample
{
    static void Main(string[] args)
    {
        int[] values = { 2, 4, 6, 8 };
        DoubleValues(values);
        foreach (var value in values)
            Console.Write("{0} ", value);
    }

    public static void DoubleValues(int[] arr)
    {
        for (int ctr = 0; ctr <= arr.GetUpperBound(0); ctr++)
            arr[ctr] = arr[ctr] * 2;
    }
}

// The example displays the following output:
//      4 8 12 16
```


Extension methods

Ordinarily, there are two ways to add a method to an existing type:

- Modify the source code for that type. You cannot do this, of course, if you do not own the type's source code. And this becomes a breaking change if you also add any private data fields to support the method.
- Define the new method in a derived class. A method cannot be added in this way using inheritance for other types, such as structures and enumerations. Nor can it be used to "add" a method to a sealed class.

Extension methods let you "add" a method to an existing type without modifying the type itself or implementing the new method in an inherited type. The extension method also does not have to reside in the same assembly as the type it extends. You call an extension method as if it were a defined member of a type.

For more information, see [Extension Methods](#).

Async Methods

By using the async feature, you can invoke asynchronous methods without using explicit callbacks or manually splitting your code across multiple methods or lambda expressions.

If you mark a method with the `async` modifier, you can use the `await` operator in the method. When control reaches an `await` expression in the async method, control returns to the caller if the awaited task is not completed, and progress in the method with the `await` keyword is suspended until the awaited task completes. When the task is complete, execution can resume in the method.

NOTE

An async method returns to the caller when either it encounters the first awaited object that's not yet complete or it gets to the end of the async method, whichever occurs first.

An async method typically has a return type of `Task<TResult>`, `Task`, `IAsyncEnumerable<T>` or `void`. The `void` return type is used primarily to define event handlers, where a `void` return type is required. An async method that returns `void` can't be awaited, and the caller of a void-returning method can't catch exceptions that the method throws. Starting with C# 7.0, an async method can have [any task-like return type](#).

In the following example, `DelayAsync` is an async method that has a return statement that returns an integer. Because it is an async method, its method declaration must have a return type of `Task<int>`. Because the return type is `Task<int>`, the evaluation of the `await` expression in `DoSomethingAsync` produces an integer, as the following `int result = await delayTask` statement demonstrates.

```

using System;
using System.Threading.Tasks;

class Program
{
    static Task Main() => DoSomethingAsync();

    static async Task DoSomethingAsync()
    {
        Task<int> delayTask = DelayAsync();
        int result = await delayTask;

        // The previous two statements may be combined into
        // the following statement.
        //int result = await DelayAsync();

        Console.WriteLine($"Result: {result}");
    }

    static async Task<int> DelayAsync()
    {
        await Task.Delay(100);
        return 5;
    }
}
// Example output:
// Result: 5

```

An async method can't declare any [in](#), [ref](#), or [out](#) parameters, but it can call methods that have such parameters.

For more information about async methods, see [Asynchronous programming with async and await](#) and [Async return types](#).

Expression-bodied members

It is common to have method definitions that simply return immediately with the result of an expression, or that have a single statement as the body of the method. There is a syntax shortcut for defining such methods using

`=>` :

```

public Point Move(int dx, int dy) => new Point(x + dx, y + dy);
public void Print() => Console.WriteLine(First + " " + Last);
// Works with operators, properties, and indexers too.
public static Complex operator +(Complex a, Complex b) => a.Add(b);
public string Name => First + " " + Last;
public Customer this[long id] => store.LookupCustomer(id);

```

If the method returns `void` or is an async method, the body of the method must be a statement expression (same as with lambdas). For properties and indexers, they must be read-only, and you do not use the `get` accessor keyword.

Iterators

An iterator performs a custom iteration over a collection, such as a list or an array. An iterator uses the [yield return](#) statement to return each element one at a time. When a `yield return` statement is reached, the current location is remembered so that the caller can request the next element in the sequence.

The return type of an iterator can be [IEnumerable](#), [IEnumerable<T>](#), [IEnumerator](#), or [IEnumerator<T>](#).

For more information, see [Iterators](#).

See also

- [Access Modifiers](#)
- [Static Classes and Static Class Members](#)
- [Inheritance](#)
- [Abstract and Sealed Classes and Class Members](#)
- [params](#)
- [out](#)
- [ref](#)
- [in](#)
- [Passing Parameters](#)

Properties

12/28/2021 • 9 minutes to read • [Edit Online](#)

Properties are first class citizens in C#. The language defines syntax that enables developers to write code that accurately expresses their design intent.

Properties behave like fields when they are accessed. However, unlike fields, properties are implemented with accessors that define the statements executed when a property is accessed or assigned.

Property syntax

The syntax for properties is a natural extension to fields. A field defines a storage location:

```
public class Person
{
    public string FirstName;
    // remaining implementation removed from listing
}
```

A property definition contains declarations for a `get` and `set` accessor that retrieves and assigns the value of that property:

```
public class Person
{
    public string FirstName { get; set; }

    // remaining implementation removed from listing
}
```

The syntax shown above is the *auto property* syntax. The compiler generates the storage location for the field that backs up the property. The compiler also implements the body of the `get` and `set` accessors.

Sometimes, you need to initialize a property to a value other than the default for its type. C# enables that by setting a value after the closing brace for the property. You may prefer the initial value for the `FirstName` property to be the empty string rather than `null`. You would specify that as shown below:

```
public class Person
{
    public string FirstName { get; set; } = string.Empty;

    // remaining implementation removed from listing
}
```

Specific initialization is most useful for read-only properties, as you'll see later in this article.

You can also define the storage yourself, as shown below:

```
public class Person
{
    public string FirstName
    {
        get { return firstName; }
        set { firstName = value; }
    }
    private string firstName;
    // remaining implementation removed from listing
}
```

When a property implementation is a single expression, you can use *expression-bodied members* for the getter or setter:

```
public class Person
{
    public string FirstName
    {
        get => firstName;
        set => firstName = value;
    }
    private string firstName;
    // remaining implementation removed from listing
}
```

This simplified syntax will be used where applicable throughout this article.

The property definition shown above is a read-write property. Notice the keyword `value` in the set accessor. The `set` accessor always has a single parameter named `value`. The `get` accessor must return a value that is convertible to the type of the property (`string` in this example).

That's the basics of the syntax. There are many different variations that support a variety of different design idioms. Let's explore, and learn the syntax options for each.

Scenarios

The examples above showed one of the simplest cases of property definition: a read-write property with no validation. By writing the code you want in the `get` and `set` accessors, you can create many different scenarios.

Validation

You can write code in the `set` accessor to ensure that the values represented by a property are always valid. For example, suppose one rule for the `Person` class is that the name cannot be blank or white space. You would write that as follows:

```

public class Person
{
    public string FirstName
    {
        get => firstName;
        set
        {
            if (string.IsNullOrEmpty(value))
                throw new ArgumentException("First name must not be blank");
            firstName = value;
        }
    }
    private string firstName;
    // remaining implementation removed from listing
}

```

The preceding example can be simplified by using a `throw` expression as part of the property setter validation:

```

public class Person
{
    public string FirstName
    {
        get => firstName;
        set => firstName = (!string.IsNullOrEmpty(value)) ? value : throw new ArgumentException("First
name must not be blank");
    }
    private string firstName;
    // remaining implementation removed from listing
}

```

The example above enforces the rule that the first name must not be blank or white space. If a developer writes

```

hero.FirstName = "";

```

That assignment throws an `ArgumentException`. Because a property set accessor must have a void return type, you report errors in the set accessor by throwing an exception.

You can extend this same syntax to anything needed in your scenario. You can check the relationships between different properties, or validate against any external conditions. Any valid C# statements are valid in a property accessor.

Read-only

Up to this point, all the property definitions you have seen are read/write properties with public accessors. That's not the only valid accessibility for properties. You can create read-only properties, or give different accessibility to the set and get accessors. Suppose that your `Person` class should only enable changing the value of the `FirstName` property from other methods in that class. You could give the set accessor `private` accessibility instead of `public`:

```

public class Person
{
    public string FirstName { get; private set; }

    // remaining implementation removed from listing
}

```

Now, the `FirstName` property can be accessed from any code, but it can only be assigned from other code in the `Person` class.

You can add any restrictive access modifier to either the set or get accessors. Any access modifier you place on the individual accessor must be more limited than the access modifier on the property definition. The above is legal because the `FirstName` property is `public`, but the set accessor is `private`. You could not declare a `private` property with a `public` accessor. Property declarations can also be declared `protected`, `internal`, `protected internal`, or, even `private`.

It is also legal to place the more restrictive modifier on the `get` accessor. For example, you could have a `public` property, but restrict the `get` accessor to `private`. That scenario is rarely done in practice.

You can also restrict modifications to a property so that it can only be set in a constructor or a property initializer. You can modify the `Person` class so as follows:

```
public class Person
{
    public Person(string firstName) => this.FirstName = firstName;

    public string FirstName { get; }

    // remaining implementation removed from listing
}
```

This feature is most commonly used for initializing collections that are exposed as read-only properties:

```
public class Measurements
{
    public ICollection<DataPoint> points { get; } = new List<DataPoint>();
}
```

Computed properties

A property does not need to simply return the value of a member field. You can create properties that return a computed value. Let's expand the `Person` object to return the full name, computed by concatenating the first and last names:

```
public class Person
{
    public string FirstName { get; set; }

    public string LastName { get; set; }

    public string FullName { get { return $"{FirstName} {LastName}"; } }
}
```

The example above uses the [string interpolation](#) feature to create the formatted string for the full name.

You can also use an *expression-bodied member*, which provides a more succinct way to create the computed `FullName` property:

```
public class Person
{
    public string FirstName { get; set; }

    public string LastName { get; set; }

    public string FullName => $"{FirstName} {LastName}";
}
```

Expression-bodied members use the *lambda expression* syntax to define methods that contain a single

expression. Here, that expression returns the full name for the person object.

Cached evaluated properties

You can mix the concept of a computed property with storage and create a *cached evaluated property*. For example, you could update the `FullName` property so that the string formatting only happened the first time it was accessed:

```
public class Person
{
    public string FirstName { get; set; }

    public string LastName { get; set; }

    private string fullName;
    public string FullName
    {
        get
        {
            if (fullName == null)
                fullName = $"{FirstName} {LastName}";
            return fullName;
        }
    }
}
```

The above code contains a bug though. If code updates the value of either the `FirstName` or `LastName` property, the previously evaluated `fullName` field is invalid. You modify the `set` accessors of the `FirstName` and `LastName` property so that the `fullName` field is calculated again:


```

public class Person
{
    private string firstName;
    public string FirstName
    {
        get => firstName;
        set
        {
            firstName = value;
            fullName = null;
        }
    }

    private string lastName;
    public string LastName
    {
        get => lastName;
        set
        {
            lastName = value;
            fullName = null;
        }
    }

    private string fullName;
    public string FullName
    {
        get
        {
            if (fullName == null)
                fullName = $"{FirstName} {LastName}";
            return fullName;
        }
    }
}

```

This final version evaluates the `FullName` property only when needed. If the previously calculated version is valid, it's used. If another state change invalidates the previously calculated version, it will be recalculated. Developers that use this class do not need to know the details of the implementation. None of these internal changes affect the use of the `Person` object. That's the key reason for using Properties to expose data members of an object.

Attaching attributes to auto-implemented properties

Beginning with C# 7.3, field attributes can be attached to the compiler generated backing field in auto-implemented properties. For example, consider a revision to the `Person` class that adds a unique integer `Id` property. You write the `Id` property using an auto-implemented property, but your design does not call for persisting the `Id` property. The `NonSerializedAttribute` can only be attached to fields, not properties. You can attach the `NonSerializedAttribute` to the backing field for the `Id` property by using the `field:` specifier on the attribute, as shown in the following example:

```

public class Person
{
    public string FirstName { get; set; }

    public string LastName { get; set; }

    [field:NonSerialized]
    public int Id { get; set; }

    public string FullName => $"{FirstName} {LastName}";
}

```

This technique works for any attribute you attach to the backing field on the auto-implemented property.

Implementing INotifyPropertyChanged

A final scenario where you need to write code in a property accessor is to support the [INotifyPropertyChanged](#) interface used to notify data binding clients that a value has changed. When the value of a property changes, the object raises the [INotifyPropertyChanged.PropertyChanged](#) event to indicate the change. The data binding libraries, in turn, update display elements based on that change. The code below shows how you would implement `INotifyPropertyChanged` for the `FirstName` property of this person class.

```
public class Person : INotifyPropertyChanged
{
    public string FirstName
    {
        get => firstName;
        set
        {
            if (string.IsNullOrEmpty(value))
                throw new ArgumentException("First name must not be blank");
            if (value != firstName)
            {
                firstName = value;
                PropertyChanged?.Invoke(this,
                    new PropertyChangedEventArgs(nameof(FirstName)));
            }
        }
    }
    private string firstName;

    public event PropertyChangedEventHandler PropertyChanged;
    // remaining implementation removed from listing
}
```

The `?.` operator is called the *null conditional operator*. It checks for a null reference before evaluating the right side of the operator. The end result is that if there are no subscribers to the `PropertyChanged` event, the code to raise the event doesn't execute. It would throw a `NullReferenceException` without this check in that case. For more information, see [events](#). This example also uses the new `nameof` operator to convert from the property name symbol to its text representation. Using `nameof` can reduce errors where you have mistyped the name of the property.

Again, implementing [INotifyPropertyChanged](#) is an example of a case where you can write code in your accessors to support the scenarios you need.

Summing up

Properties are a form of smart fields in a class or object. From outside the object, they appear like fields in the object. However, properties can be implemented using the full palette of C# functionality. You can provide validation, different accessibility, lazy evaluation, or any requirements your scenarios need.

Indexers

12/28/2021 • 9 minutes to read • [Edit Online](#)

Indexers are similar to properties. In many ways indexers build on the same language features as [properties](#). Indexers enable *indexed* properties: properties referenced using one or more arguments. Those arguments provide an index into some collection of values.

Indexer Syntax

You access an indexer through a variable name and square brackets. You place the indexer arguments inside the brackets:

```
var item = someObject["key"];
someObject["AnotherKey"] = item;
```

You declare indexers using the `this` keyword as the property name, and declaring the arguments within square brackets. This declaration would match the usage shown in the previous paragraph:

```
public int this[string key]
{
    get { return storage.Find(key); }
    set { storage.SetAt(key, value); }
}
```

From this initial example, you can see the relationship between the syntax for properties and for indexers. This analogy carries through most of the syntax rules for indexers. Indexers can have any valid access modifiers (public, protected internal, protected, internal, private or private protected). They may be sealed, virtual, or abstract. As with properties, you can specify different access modifiers for the get and set accessors in an indexer. You may also specify read-only indexers (by omitting the set accessor), or write-only indexers (by omitting the get accessor).

You can apply almost everything you learn from working with properties to indexers. The only exception to that rule is *auto implemented properties*. The compiler cannot always generate the correct storage for an indexer.

The presence of arguments to reference an item in a set of items distinguishes indexers from properties. You may define multiple indexers on a type, as long as the argument lists for each indexer is unique. Let's explore different scenarios where you might use one or more indexers in a class definition.

Scenarios

You would define *indexers* in your type when its API models some collection where you define the arguments to that collection. Your indexers may or may not map directly to the collection types that are part of the .NET core framework. Your type may have other responsibilities in addition to modeling a collection. Indexers enable you to provide the API that matches your type's abstraction without exposing the inner details of how the values for that abstraction are stored or computed.

Let's walk through some of the common scenarios for using *indexers*. You can access the [sample folder for indexers](#). For download instructions, see [Samples and Tutorials](#).

Arrays and Vectors

One of the most common scenarios for creating indexers is when your type models an array, or a vector. You can

create an indexer to model an ordered list of data.

The advantage of creating your own indexer is that you can define the storage for that collection to suit your needs. Imagine a scenario where your type models historical data that is too large to load into memory at once. You need to load and unload sections of the collection based on usage. The example following models this behavior. It reports on how many data points exist. It creates pages to hold sections of the data on demand. It removes pages from memory to make room for pages needed by more recent requests.

```
public class DataSamples
{
    private class Page
    {
        private readonly List<Measurements> pageData = new List<Measurements>();
        private readonly int startingIndex;
        private readonly int length;
        private bool dirty;
        private DateTime lastAccess;

        public Page(int startingIndex, int length)
        {
            this.startingIndex = startingIndex;
            this.length = length;
            lastAccess = DateTime.Now;

            // This stays as random stuff:
            var generator = new Random();
            for(int i=0; i < length; i++)
            {
                var m = new Measurements
                {
                    HiTemp = generator.Next(50, 95),
                    LoTemp = generator.Next(12, 49),
                    AirPressure = 28.0 + generator.NextDouble() * 4
                };
                pageData.Add(m);
            }
        }

        public bool HasItem(int index) =>
            ((index >= startingIndex) &&
            (index < startingIndex + length));

        public Measurements this[int index]
        {
            get
            {
                lastAccess = DateTime.Now;
                return pageData[index - startingIndex];
            }
            set
            {
                pageData[index - startingIndex] = value;
                dirty = true;
                lastAccess = DateTime.Now;
            }
        }
    }

    public bool Dirty => dirty;
    public DateTime LastAccess => lastAccess;
}

private readonly int totalSize;
private readonly List<Page> pagesInMemory = new List<Page>();

public DataSamples(int totalSize)
{
    this.totalSize = totalSize;
}
```

```

    }

    public Measurements this[int index]
    {
        get
        {
            if (index < 0)
                throw new IndexOutOfRangeException("Cannot index less than 0");
            if (index >= totalSize)
                throw new IndexOutOfRangeException("Cannot index past the end of storage");

            var page = updateCachedPagesForAccess(index);
            return page[index];
        }
        set
        {
            if (index < 0)
                throw new IndexOutOfRangeException("Cannot index less than 0");
            if (index >= totalSize)
                throw new IndexOutOfRangeException("Cannot index past the end of storage");
            var page = updateCachedPagesForAccess(index);

            page[index] = value;
        }
    }

    private Page updateCachedPagesForAccess(int index)
    {
        foreach (var p in pagesInMemory)
        {
            if (p.HasItem(index))
            {
                return p;
            }
        }
        var startingIndex = (index / 1000) * 1000;
        var newPage = new Page(startingIndex, 1000);
        addPageToCache(newPage);
        return newPage;
    }

    private void addPageToCache(Page p)
    {
        if (pagesInMemory.Count > 4)
        {
            // remove oldest non-dirty page:
            var oldest = pagesInMemory
                .Where(page => !page.Dirty)
                .OrderBy(page => page.LastAccess)
                .FirstOrDefault();
            // Note that this may keep more than 5 pages in memory
            // if too much is dirty
            if (oldest != null)
                pagesInMemory.Remove(oldest);
        }
        pagesInMemory.Add(p);
    }
}

```

You can follow this design idiom to model any sort of collection where there are good reasons not to load the entire set of data into an in-memory collection. Notice that the `Page` class is a private nested class that is not part of the public interface. Those details are hidden from any users of this class.

Dictionaries

Another common scenario is when you need to model a dictionary or a map. This scenario is when your type stores values based on key, typically text keys. This example creates a dictionary that maps command line

arguments to [lambda expressions](#) that manage those options. The following example shows two classes: an `ArgsActions` class that maps a command line option to an `Action` delegate, and an `ArgsProcessor` that uses the `ArgsActions` to execute each `Action` when it encounters that option.

```
public class ArgsProcessor
{
    private readonly ArgsActions actions;

    public ArgsProcessor(ArgsActions actions)
    {
        this.actions = actions;
    }

    public void Process(string[] args)
    {
        foreach(var arg in args)
        {
            actions[arg]?.Invoke();
        }
    }
}

public class ArgsActions
{
    readonly private Dictionary<string, Action> argsActions = new Dictionary<string, Action>();

    public Action this[string s]
    {
        get
        {
            Action action;
            Action defaultAction = () => {} ;
            return argsActions.TryGetValue(s, out action) ? action : defaultAction;
        }
    }

    public void SetOption(string s, Action a)
    {
        argsActions[s] = a;
    }
}
```

In this example, the `ArgsAction` collection maps closely to the underlying collection. The `get` determines if a given option has been configured. If so, it returns the `Action` associated with that option. If not, it returns an `Action` that does nothing. The public accessor does not include a `set` accessor. Rather, the design is using a public method for setting options.

Multi-Dimensional Maps

You can create indexers that use multiple arguments. In addition, those arguments are not constrained to be the same type. Let's look at two examples.

The first example shows a class that generates values for a Mandelbrot set. For more information on the mathematics behind the set, read [this article](#). The indexer uses two doubles to define a point in the X, Y plane. The `get` accessor computes the number of iterations until a point is determined to be not in the set. If the maximum iterations is reached, the point is in the set, and the class's `maxIterations` value is returned. (The computer generated images popularized for the Mandelbrot set define colors for the number of iterations necessary to determine that a point is outside the set.)

```

public class Mandelbrot
{
    readonly private int maxIterations;

    public Mandelbrot(int maxIterations)
    {
        this.maxIterations = maxIterations;
    }

    public int this [double x, double y]
    {
        get
        {
            var iterations = 0;
            var x0 = x;
            var y0 = y;

            while ((x*x + y * y < 4) &&
                (iterations < maxIterations))
            {
                var newX = x * x - y * y + x0;
                y = 2 * x * y + y0;
                x = newX;
                iterations++;
            }
            return iterations;
        }
    }
}

```

The Mandelbrot Set defines values at every (x,y) coordinate for real number values. That defines a dictionary that could contain an infinite number of values. Therefore, there is no storage behind the set. Instead, this class computes the value for each point when code calls the `get` accessor. There's no underlying storage used.

Let's examine one last use of indexers, where the indexer takes multiple arguments of different types. Consider a program that manages historical temperature data. This indexer uses a city and a date to set or get the high and low temperatures for that location:

```

using DateMeasurements =
    System.Collections.Generic.Dictionary<System.DateTime, IndexersSamples.Common.Measurements>;
using CityDataMeasurements =
    System.Collections.Generic.Dictionary<string, System.Collections.Generic.Dictionary<System.DateTime,
IndexersSamples.Common.Measurements>>;

public class HistoricalWeatherData
{
    readonly CityDataMeasurements storage = new CityDataMeasurements();

    public Measurements this[string city, DateTime date]
    {
        get
        {
            var cityData = default(DateMeasurements);

            if (!storage.TryGetValue(city, out cityData))
                throw new ArgumentOutOfRangeException(nameof(city), "City not found");

            // strip out any time portion:
            var index = date.Date;
            var measure = default(Measurements);
            if (cityData.TryGetValue(index, out measure))
                return measure;
            throw new ArgumentOutOfRangeException(nameof(date), "Date not found");
        }
        set
        {
            var cityData = default(DateMeasurements);

            if (!storage.TryGetValue(city, out cityData))
            {
                cityData = new DateMeasurements();
                storage.Add(city, cityData);
            }

            // Strip out any time portion:
            var index = date.Date;
            cityData[index] = value;
        }
    }
}

```

This example creates an indexer that maps weather data on two different arguments: a city (represented by a `string`) and a date (represented by a `DateTime`). The internal storage uses two `Dictionary` classes to represent the two-dimensional dictionary. The public API no longer represents the underlying storage. Rather, the language features of indexers enables you to create a public interface that represents your abstraction, even though the underlying storage must use different core collection types.

There are two parts of this code that may be unfamiliar to some developers. These two `using` directives:

```

using DateMeasurements = System.Collections.Generic.Dictionary<System.DateTime,
IndexersSamples.Common.Measurements>;
using CityDataMeasurements = System.Collections.Generic.Dictionary<string,
System.Collections.Generic.Dictionary<System.DateTime, IndexersSamples.Common.Measurements>>;

```

create an *alias* for a constructed generic type. Those statements enable the code later to use the more descriptive `DateMeasurements` and `CityDataMeasurements` names instead of the generic construction of `Dictionary<DateTime, Measurements>` and `Dictionary<string, Dictionary<DateTime, Measurements>>`. This construct does require using the fully qualified type names on the right side of the `=` sign.

The second technique is to strip off the time portions of any `DateTime` object used to index into the collections.

.NET doesn't include a date-only type. Developers use the `DateTime` type, but use the `Date` property to ensure that any `DateTime` object from that day are equal.

Summing Up

You should create indexers anytime you have a property-like element in your class where that property represents not a single value, but rather a collection of values where each individual item is identified by a set of arguments. Those arguments can uniquely identify which item in the collection should be referenced. Indexers extend the concept of [properties](#), where a member is treated like a data item from outside the class, but like a method on the inside. Indexers allow arguments to find a single item in a property that represents a set of items.

Iterators

12/28/2021 • 6 minutes to read • [Edit Online](#)

Almost every program you write will have some need to iterate over a collection. You'll write code that examines every item in a collection.

You'll also create iterator methods, which are methods that produce an *iterator* for the elements of that class. An *iterator* is an object that traverses a container, particularly lists. Iterators can be used for:

- Performing an action on each item in a collection.
- Enumerating a custom collection.
- Extending [LINQ](#) or other libraries.
- Creating a data pipeline where data flows efficiently through iterator methods.

The C# language provides features for both generating and consuming sequences. These sequences can be produced and consumed synchronously or asynchronously. This article provides an overview of those features.

Iterating with foreach

Enumerating a collection is simple: The `foreach` keyword enumerates a collection, executing the embedded statement once for each element in the collection:

```
foreach (var item in collection)
{
    Console.WriteLine(item.ToString());
}
```

That's all. To iterate over all the contents of a collection, the `foreach` statement is all you need. The `foreach` statement isn't magic, though. It relies on two generic interfaces defined in the .NET core library to generate the code necessary to iterate a collection: `IEnumerable<T>` and `IEnumerator<T>`. This mechanism is explained in more detail below.

Both of these interfaces also have non-generic counterparts: `IEnumerable` and `IEnumerator`. The [generic](#) versions are preferred for modern code.

When a sequence is generated asynchronously, you can use the `await foreach` statement to asynchronously consume the sequence:

```
await foreach (var item in asyncSequence)
{
    Console.WriteLine(item.ToString());
}
```

When a sequence is an [System.Collections.Generic.IEnumerable<T>](#), you use `foreach`. When a sequence is an [System.Collections.Generic.IAsyncEnumerable<T>](#), you use `await foreach`. In the latter case, the sequence is generated asynchronously.

Enumeration sources with iterator methods

Another great feature of the C# language enables you to build methods that create a source for an enumeration. These methods are referred to as *iterator methods*. An iterator method defines how to generate the objects in a

sequence when requested. You use the `yield return` contextual keywords to define an iterator method.

You could write this method to produce the sequence of integers from 0 through 9:

```
public IEnumerable<int> GetSingleDigitNumbers()
{
    yield return 0;
    yield return 1;
    yield return 2;
    yield return 3;
    yield return 4;
    yield return 5;
    yield return 6;
    yield return 7;
    yield return 8;
    yield return 9;
}
```

The code above shows distinct `yield return` statements to highlight the fact that you can use multiple discrete `yield return` statements in an iterator method. You can (and often do) use other language constructs to simplify the code of an iterator method. The method definition below produces the exact same sequence of numbers:

```
public IEnumerable<int> GetSingleDigitNumbersLoop()
{
    int index = 0;
    while (index < 10)
        yield return index++;
}
```

You don't have to decide one or the other. You can have as many `yield return` statements as necessary to meet the needs of your method:

```
public IEnumerable<int> GetSetsOfNumbers()
{
    int index = 0;
    while (index < 10)
        yield return index++;

    yield return 50;

    index = 100;
    while (index < 110)
        yield return index++;
}
```

All of these preceding examples would have an asynchronous counterpart. In each case, you'd replace the return type of `IEnumerable<T>` with an `IAsyncEnumerable<T>`. For example, the previous example would have the following asynchronous version:

```

public async IAsyncEnumerable<int> GetSetsOfNumbersAsync()
{
    int index = 0;
    while (index < 10)
        yield return index++;

    await Task.Delay(500);

    yield return 50;

    await Task.Delay(500);

    index = 100;
    while (index < 110)
        yield return index++;
}

```

That's the syntax for both synchronous and asynchronous iterators. Let's consider a real world example. Imagine you're on an IoT project and the device sensors generate a very large stream of data. To get a feel for the data, you might write a method that samples every Nth data element. This small iterator method does the trick:

```

public static IEnumerable<T> Sample<T>(this IEnumerable<T> sourceSequence, int interval)
{
    int index = 0;
    foreach (T item in sourceSequence)
    {
        if (index++ % interval == 0)
            yield return item;
    }
}

```

If reading from the IoT device produces an asynchronous sequence, you'd modify the method as the following method shows:

```

public static async IAsyncEnumerable<T> Sample<T>(this IAsyncEnumerable<T> sourceSequence, int interval)
{
    int index = 0;
    await foreach (T item in sourceSequence)
    {
        if (index++ % interval == 0)
            yield return item;
    }
}

```

There's one important restriction on iterator methods: you can't have both a `return` statement and a `yield return` statement in the same method. The following code won't compile:

```

public IEnumerable<int> GetSingleDigitNumbers()
{
    int index = 0;
    while (index < 10)
        yield return index++;

    yield return 50;

    // generates a compile time error:
    var items = new int[] {100, 101, 102, 103, 104, 105, 106, 107, 108, 109 };
    return items;
}

```

This restriction normally isn't a problem. You have a choice of either using `yield return` throughout the method, or separating the original method into multiple methods, some using `return`, and some using `yield return`.

You can modify the last method slightly to use `yield return` everywhere:

```
public IEnumerable<int> GetFirstDecile()
{
    int index = 0;
    while (index < 10)
        yield return index++;

    yield return 50;

    var items = new int[] {100, 101, 102, 103, 104, 105, 106, 107, 108, 109 };
    foreach (var item in items)
        yield return item;
}
```

Sometimes, the right answer is to split an iterator method into two different methods. One that uses `return`, and a second that uses `yield return`. Consider a situation where you might want to return an empty collection, or the first five odd numbers, based on a boolean argument. You could write that as these two methods:

```
public IEnumerable<int> GetSingleDigitOddNumbers(bool getCollection)
{
    if (getCollection == false)
        return new int[0];
    else
        return IteratorMethod();
}

private IEnumerable<int> IteratorMethod()
{
    int index = 0;
    while (index < 10)
    {
        if (index % 2 == 1)
            yield return index;
        index++;
    }
}
```

Look at the methods above. The first uses the standard `return` statement to return either an empty collection, or the iterator created by the second method. The second method uses the `yield return` statement to create the requested sequence.

Deeper dive into `foreach`

The `foreach` statement expands into a standard idiom that uses the `IEnumerable<T>` and `IEnumerator<T>` interfaces to iterate across all elements of a collection. It also minimizes errors developers make by not properly managing resources.

The compiler translates the `foreach` loop shown in the first example into something similar to this construct:

```
IEnumerator<int> enumerator = collection.GetEnumerator();
while (enumerator.MoveNext())
{
    var item = enumerator.Current;
    Console.WriteLine(item.ToString());
}
```

The exact code generated by the compiler is more complicated, and handles situations where the object returned by `GetEnumerator()` implements the `IDisposable` interface. The full expansion generates code more like this:

```
{
    var enumerator = collection.GetEnumerator();
    try
    {
        while (enumerator.MoveNext())
        {
            var item = enumerator.Current;
            Console.WriteLine(item.ToString());
        }
    }
    finally
    {
        // dispose of enumerator.
    }
}
```

The compiler translates the first asynchronous sample into something similar to this construct:

```
{
    var enumerator = collection.GetAsyncEnumerator();
    try
    {
        while (await enumerator.MoveNextAsync())
        {
            var item = enumerator.Current;
            Console.WriteLine(item.ToString());
        }
    }
    finally
    {
        // dispose of async enumerator.
    }
}
```

The manner in which the enumerator is disposed of depends on the characteristics of the type of `enumerator`. In the general synchronous case, the `finally` clause expands to:

```
finally
{
    (enumerator as IDisposable)?.Dispose();
}
```

The general asynchronous case expands to:

```
finally
{
    if (enumerator is IAsyncDisposable asyncDisposable)
        await asyncDisposable.DisposeAsync();
}
```

However, if the type of `enumerator` is a sealed type and there's no implicit conversion from the type of `enumerator` to `IDisposable` or `IAsyncDisposable`, the `finally` clause expands to an empty block:

```
finally
{
}
```

If there's an implicit conversion from the type of `enumerator` to `IDisposable`, and `enumerator` is a non-nullable value type, the `finally` clause expands to:

```
finally
{
    ((IDisposable)enumerator).Dispose();
}
```

Thankfully, you don't need to remember all these details. The `foreach` statement handles all those nuances for you. The compiler will generate the correct code for any of these constructs.

Introduction to Delegates

12/28/2021 • 2 minutes to read • [Edit Online](#)

Delegates provide a *late binding* mechanism in .NET. Late Binding means that you create an algorithm where the caller also supplies at least one method that implements part of the algorithm.

For example, consider sorting a list of stars in an astronomy application. You may choose to sort those stars by their distance from the earth, or the magnitude of the star, or their perceived brightness.

In all those cases, the Sort() method does essentially the same thing: arranges the items in the list based on some comparison. The code that compares two stars is different for each of the sort orderings.

These kinds of solutions have been used in software for half a century. The C# language delegate concept provides first class language support, and type safety around the concept.

As you'll see later in this series, the C# code you write for algorithms like this is type safe, and uses the language rules and the compiler to ensure that the types match for arguments and return types.

[Function pointers](#) were added to C# 9 for similar scenarios, where you need more control over the calling convention. The code associated with a delegate is invoked using a virtual method added to a delegate type. Using function pointers, you can specify different conventions.

Language Design Goals for Delegates

The language designers enumerated several goals for the feature that eventually became delegates.

The team wanted a common language construct that could be used for any late binding algorithms. Delegates enable developers to learn one concept, and use that same concept across many different software problems.

Second, the team wanted to support both single and multicast method calls. (Multicast delegates are delegates that chain together multiple method calls. You'll see examples [later in this series](#).)

The team wanted delegates to support the same type safety that developers expect from all C# constructs.

Finally, the team recognized an event pattern is one specific pattern where delegates, or any late binding algorithm, is very useful. The team wanted to ensure the code for delegates could provide the basis for the .NET event pattern.

The result of all that work was the delegate and event support in C# and .NET. The remaining articles in this section will cover language features, library support, and common idioms used when you work with delegates.

You'll learn about the `delegate` keyword and what code it generates. You'll learn about the features in the `System.Delegate` class, and how those features are used. You'll learn how to create type safe delegates, and how to create methods that can be invoked through delegates. You'll also learn how to work with delegates and events by using Lambda expressions. You'll see where delegates become one of the building blocks for LINQ. You'll learn how delegates are the basis for the .NET event pattern, and how they're different.

Let's get started.

[Next](#)

System.Delegate and the `delegate` keyword

12/28/2021 • 6 minutes to read • [Edit Online](#)

[Previous](#)

This article covers the classes in .NET that support delegates, and how those map to the `delegate` keyword.

Define delegate types

Let's start with the 'delegate' keyword, because that's primarily what you will use as you work with delegates. The code that the compiler generates when you use the `delegate` keyword will map to method calls that invoke members of the [Delegate](#) and [MulticastDelegate](#) classes.

You define a delegate type using syntax that is similar to defining a method signature. You just add the `delegate` keyword to the definition.

Let's continue to use the `List.Sort()` method as our example. The first step is to create a type for the comparison delegate:

```
// From the .NET Core library

// Define the delegate type:
public delegate int Comparison<in T>(T left, T right);
```

The compiler generates a class, derived from `System.Delegate` that matches the signature used (in this case, a method that returns an integer, and has two arguments). The type of that delegate is `Comparison`. The `Comparison` delegate type is a generic type. For details on generics see [here](#).

Notice that the syntax may appear as though it is declaring a variable, but it is actually declaring a *type*. You can define delegate types inside classes, directly inside namespaces, or even in the global namespace.

NOTE

Declaring delegate types (or other types) directly in the global namespace is not recommended.

The compiler also generates add and remove handlers for this new type so that clients of this class can add and remove methods from an instance's invocation list. The compiler will enforce that the signature of the method being added or removed matches the signature used when declaring the method.

Declare instances of delegates

After defining the delegate, you can create an instance of that type. Like all variables in C#, you cannot declare delegate instances directly in a namespace, or in the global namespace.

```
// inside a class definition:

// Declare an instance of that type:
public Comparison<T> comparator;
```

The type of the variable is `Comparison<T>`, the delegate type defined earlier. The name of the variable is

`comparator` .

That code snippet above declared a member variable inside a class. You can also declare delegate variables that are local variables, or arguments to methods.

Invoke delegates

You invoke the methods that are in the invocation list of a delegate by calling that delegate. Inside the `Sort()` method, the code will call the comparison method to determine which order to place objects:

```
int result = comparator(left, right);
```

In the line above, the code *invokes* the method attached to the delegate. You treat the variable as a method name, and invoke it using normal method call syntax.

That line of code makes an unsafe assumption: There's no guarantee that a target has been added to the delegate. If no targets have been attached, the line above would cause a `NullReferenceException` to be thrown. The idioms used to address this problem are more complicated than a simple null-check, and are covered later in this [series](#).

Assign, add, and remove invocation targets

That's how a delegate type is defined, and how delegate instances are declared and invoked.

Developers that want to use the `List.Sort()` method need to define a method whose signature matches the delegate type definition, and assign it to the delegate used by the sort method. This assignment adds the method to the invocation list of that delegate object.

Suppose you wanted to sort a list of strings by their length. Your comparison function might be the following:

```
private static int CompareLength(string left, string right) =>
    left.Length.CompareTo(right.Length);
```

The method is declared as a private method. That's fine. You may not want this method to be part of your public interface. It can still be used as the comparison method when attached to a delegate. The calling code will have this method attached to the target list of the delegate object, and can access it through that delegate.

You create that relationship by passing that method to the `List.Sort()` method:

```
phrases.Sort(CompareLength);
```

Notice that the method name is used, without parentheses. Using the method as an argument tells the compiler to convert the method reference into a reference that can be used as a delegate invocation target, and attach that method as an invocation target.

You could also have been explicit by declaring a variable of type `Comparison<string>` and doing an assignment:

```
Comparison<string> comparer = CompareLength;
phrases.Sort(comparer);
```

In uses where the method being used as a delegate target is a small method, it's common to use [lambda expression](#) syntax to perform the assignment:

```
Comparison<string> comparer = (left, right) => left.Length.CompareTo(right.Length);
phrases.Sort(comparer);
```

Using lambda expressions for delegate targets is covered more in a [later section](#).

The Sort() example typically attaches a single target method to the delegate. However, delegate objects do support invocation lists that have multiple target methods attached to a delegate object.

Delegate and MulticastDelegate classes

The language support described above provides the features and support you'll typically need to work with delegates. These features are built on two classes in the .NET Core framework: [Delegate](#) and [MulticastDelegate](#).

The `System.Delegate` class and its single direct subclass, `System.MulticastDelegate`, provide the framework support for creating delegates, registering methods as delegate targets, and invoking all methods that are registered as a delegate target.

Interestingly, the `System.Delegate` and `System.MulticastDelegate` classes are not themselves delegate types. They do provide the basis for all specific delegate types. That same language design process mandated that you cannot declare a class that derives from `Delegate` or `MulticastDelegate`. The C# language rules prohibit it.

Instead, the C# compiler creates instances of a class derived from `MulticastDelegate` when you use the C# language keyword to declare delegate types.

This design has its roots in the first release of C# and .NET. One goal for the design team was to ensure that the language enforced type safety when using delegates. That meant ensuring that delegates were invoked with the right type and number of arguments. And, that any return type was correctly indicated at compile time. Delegates were part of the 1.0 .NET release, which was before generics.

The best way to enforce this type safety was for the compiler to create the concrete delegate classes that represented the method signature being used.

Even though you cannot create derived classes directly, you will use the methods defined on these classes. Let's go through the most common methods that you will use when you work with delegates.

The first, most important fact to remember is that every delegate you work with is derived from `MulticastDelegate`. A multicast delegate means that more than one method target can be invoked when invoking through a delegate. The original design considered making a distinction between delegates where only one target method could be attached and invoked, and delegates where multiple target methods could be attached and invoked. That distinction proved to be less useful in practice than originally thought. The two different classes were already created, and have been in the framework since its initial public release.

The methods that you will use the most with delegates are `Invoke()` and `BeginInvoke()` / `EndInvoke()`.

`Invoke()` will invoke all the methods that have been attached to a particular delegate instance. As you saw above, you typically invoke delegates using the method call syntax on the delegate variable. As you'll see [later in this series](#), there are patterns that work directly with these methods.

Now that you've seen the language syntax and the classes that support delegates, let's examine how strongly typed delegates are used, created, and invoked.

[Next](#)

Strongly Typed Delegates

12/28/2021 • 2 minutes to read • [Edit Online](#)

[Previous](#)

In the previous article, you saw that you create specific delegate types using the `delegate` keyword.

The abstract Delegate class provides the infrastructure for loose coupling and invocation. Concrete Delegate types become much more useful by embracing and enforcing type safety for the methods that are added to the invocation list for a delegate object. When you use the `delegate` keyword and define a concrete delegate type, the compiler generates those methods.

In practice, this would lead to creating new delegate types whenever you need a different method signature. This work could get tedious after a time. Every new feature requires new delegate types.

Thankfully, this isn't necessary. The .NET Core framework contains several types that you can reuse whenever you need delegate types. These are [generic](#) definitions so you can declare customizations when you need new method declarations.

The first of these types is the [Action](#) type, and several variations:

```
public delegate void Action();
public delegate void Action<in T>(T arg);
public delegate void Action<in T1, in T2>(T1 arg1, T2 arg2);
// Other variations removed for brevity.
```

The `in` modifier on the generic type argument is covered in the article on covariance.

There are variations of the `Action` delegate that contain up to 16 arguments such as [Action<T1,T2,T3,T4,T5,T6,T7,T8,T9,T10,T11,T12,T13,T14,T15,T16>](#). It's important that these definitions use different generic arguments for each of the delegate arguments: That gives you maximum flexibility. The method arguments need not be, but may be, the same type.

Use one of the `Action` types for any delegate type that has a void return type.

The framework also includes several generic delegate types that you can use for delegate types that return values:

```
public delegate TResult Func<out TResult>();
public delegate TResult Func<in T1, out TResult>(T1 arg);
public delegate TResult Func<in T1, in T2, out TResult>(T1 arg1, T2 arg2);
// Other variations removed for brevity
```

The `out` modifier on the result generic type argument is covered in the article on covariance.

There are variations of the `Func` delegate with up to 16 input arguments such as [Func<T1,T2,T3,T4,T5,T6,T7,T8,T9,T10,T11,T12,T13,T14,T15,T16,TResult>](#). The type of the result is always the last type parameter in all the `Func` declarations, by convention.

Use one of the `Func` types for any delegate type that returns a value.

There's also a specialized [Predicate<T>](#) type for a delegate that returns a test on a single value:

```
public delegate bool Predicate<in T>(T obj);
```

You may notice that for any `Predicate` type, a structurally equivalent `Func` type exists. For example:

```
Func<string, bool> TestForString;  
Predicate<string> AnotherTestForString;
```

You might think these two types are equivalent. They are not. These two variables cannot be used interchangeably. A variable of one type cannot be assigned the other type. The C# type system uses the names of the defined types, not the structure.

All these delegate type definitions in the .NET Core Library should mean that you do not need to define a new delegate type for any new feature you create that requires delegates. These generic definitions should provide all the delegate types you need under most situations. You can simply instantiate one of these types with the required type parameters. In the case of algorithms that can be made generic, these delegates can be used as generic types.

This should save time, and minimize the number of new types that you need to create in order to work with delegates.

In the next article, you'll see several common patterns for working with delegates in practice.

[Next](#)

Common patterns for delegates

12/28/2021 • 8 minutes to read • [Edit Online](#)

[Previous](#)

Delegates provide a mechanism that enables software designs involving minimal coupling between components.

One excellent example for this kind of design is LINQ. The LINQ Query Expression Pattern relies on delegates for all of its features. Consider this simple example:

```
var smallNumbers = numbers.Where(n => n < 10);
```

This filters the sequence of numbers to only those less than the value 10. The `Where` method uses a delegate that determines which elements of a sequence pass the filter. When you create a LINQ query, you supply the implementation of the delegate for this specific purpose.

The prototype for the `Where` method is:

```
public static IEnumerable<TSource> Where<TSource> (this IEnumerable<TSource> source, Func<TSource, bool> predicate);
```

This example is repeated with all the methods that are part of LINQ. They all rely on delegates for the code that manages the specific query. This API design pattern is a powerful one to learn and understand.

This simple example illustrates how delegates require very little coupling between components. You don't need to create a class that derives from a particular base class. You don't need to implement a specific interface. The only requirement is to provide the implementation of one method that is fundamental to the task at hand.

Build Your Own Components with Delegates

Let's build on that example by creating a component using a design that relies on delegates.

Let's define a component that could be used for log messages in a large system. The library components could be used in many different environments, on multiple different platforms. There are a lot of common features in the component that manages the logs. It will need to accept messages from any component in the system. Those messages will have different priorities, which the core component can manage. The messages should have timestamps in their final archived form. For more advanced scenarios, you could filter messages by the source component.

There is one aspect of the feature that will change often: where messages are written. In some environments, they may be written to the error console. In others, a file. Other possibilities include database storage, OS event logs, or other document storage.

There are also combinations of output that might be used in different scenarios. You may want to write messages to the console and to a file.

A design based on delegates will provide a great deal of flexibility, and make it easy to support storage mechanisms that may be added in the future.

Under this design, the primary log component can be a non-virtual, even sealed class. You can plug in any set of delegates to write the messages to different storage media. The built-in support for multicast delegates makes it

easy to support scenarios where messages must be written to multiple locations (a file, and a console).

A First Implementation

Let's start small: the initial implementation will accept new messages, and write them using any attached delegate. You can start with one delegate that writes messages to the console.

```
public static class Logger
{
    public static Action<string> WriteMessage;

    public static void LogMessage(string msg)
    {
        WriteMessage(msg);
    }
}
```

The static class above is the simplest thing that can work. We need to write the single implementation for the method that writes messages to the console:

```
public static class LoggingMethods
{
    public static void LogToConsole(string message)
    {
        Console.Error.WriteLine(message);
    }
}
```

Finally, you need to hook up the delegate by attaching it to the WriteMessage delegate declared in the logger:

```
Logger.WriteMessage += LoggingMethods.LogToConsole;
```

Practices

Our sample so far is fairly simple, but it still demonstrates some of the important guidelines for designs involving delegates.

Using the delegate types defined in the core framework makes it easier for users to work with the delegates. You don't need to define new types, and developers using your library do not need to learn new, specialized delegate types.

The interfaces used are as minimal and as flexible as possible: To create a new output logger, you must create one method. That method may be a static method, or an instance method. It may have any access.

Format Output

Let's make this first version a bit more robust, and then start creating other logging mechanisms.

Next, let's add a few arguments to the `LogMessage()` method so that your log class creates more structured messages:

```
public enum Severity
{
    Verbose,
    Trace,
    Information,
    Warning,
    Error,
    Critical
}
```

```
public static class Logger
{
    public static Action<string> WriteMessage;

    public static void LogMessage(Severity s, string component, string msg)
    {
        var outputMsg = $"{DateTime.Now}\t{s}\t{component}\t{msg}";
        WriteMessage(outputMsg);
    }
}
```

Next, let's make use of that `Severity` argument to filter the messages that are sent to the log's output.

```
public static class Logger
{
    public static Action<string> WriteMessage;

    public static Severity LogLevel {get;set;} = Severity.Warning;

    public static void LogMessage(Severity s, string component, string msg)
    {
        if (s < LogLevel)
            return;

        var outputMsg = $"{DateTime.Now}\t{s}\t{component}\t{msg}";
        WriteMessage(outputMsg);
    }
}
```

Practices

You've added new features to the logging infrastructure. Because the logger component is very loosely coupled to any output mechanism, these new features can be added with no impact on any of the code implementing the logger delegate.

As you keep building this, you'll see more examples of how this loose coupling enables greater flexibility in updating parts of the site without any changes to other locations. In fact, in a larger application, the logger output classes might be in a different assembly, and not even need to be rebuilt.

Build a Second Output Engine

The Log component is coming along well. Let's add one more output engine that logs messages to a file. This will be a slightly more involved output engine. It will be a class that encapsulates the file operations, and ensures that the file is always closed after each write. That ensures that all the data is flushed to disk after each message is generated.

Here is that file-based logger:


```

public class FileLogger
{
    private readonly string logPath;
    public FileLogger(string path)
    {
        logPath = path;
        Logger.WriteMessage += LogMessage;
    }

    public void DetachLog() => Logger.WriteMessage -= LogMessage;
    // make sure this can't throw.
    private void LogMessage(string msg)
    {
        try
        {
            using (var log = File.AppendText(logPath))
            {
                log.WriteLine(msg);
                log.Flush();
            }
        }
        catch (Exception)
        {
            // Hmm. We caught an exception while
            // logging. We can't really log the
            // problem (since it's the log that's failing).
            // So, while normally, catching an exception
            // and doing nothing isn't wise, it's really the
            // only reasonable option here.
        }
    }
}

```

Once you've created this class, you can instantiate it and it attaches its LogMessage method to the Logger component:

```

var file = new FileLogger("log.txt");

```

These two are not mutually exclusive. You could attach both log methods and generate messages to the console and a file:

```

var fileOutput = new FileLogger("log.txt");
Logger.WriteMessage += LoggingMethods.LogToConsole; // LoggingMethods is the static class we utilized
earlier

```

Later, even in the same application, you can remove one of the delegates without any other issues to the system:

```

Logger.WriteMessage -= LoggingMethods.LogToConsole;

```

Practices

Now, you've added a second output handler for the logging subsystem. This one needs a bit more infrastructure to correctly support the file system. The delegate is an instance method. It's also a private method. There's no need for greater accessibility because the delegate infrastructure can connect the delegates.

Second, the delegate-based design enables multiple output methods without any extra code. You don't need to build any additional infrastructure to support multiple output methods. They simply become another method on the invocation list.

Pay special attention to the code in the file logging output method. It is coded to ensure that it does not throw any exceptions. While this isn't always strictly necessary, it's often a good practice. If either of the delegate methods throws an exception, the remaining delegates that are on the invocation won't be invoked.

As a last note, the file logger must manage its resources by opening and closing the file on each log message. You could choose to keep the file open and implement `IDisposable` to close the file when you are completed. Either method has its advantages and disadvantages. Both do create a bit more coupling between the classes.

None of the code in the `Logger` class would need to be updated in order to support either scenario.

Handle Null Delegates

Finally, let's update the `LogMessage` method so that it is robust for those cases when no output mechanism is selected. The current implementation will throw a `NullReferenceException` when the `WriteMessage` delegate does not have an invocation list attached. You may prefer a design that silently continues when no methods have been attached. This is easy using the null conditional operator, combined with the `Delegate.Invoke()` method:

```
public static void LogMessage(string msg)
{
    WriteMessage?.Invoke(msg);
}
```

The null conditional operator (`?.`) short-circuits when the left operand (`WriteMessage` in this case) is null, which means no attempt is made to log a message.

You won't find the `Invoke()` method listed in the documentation for `System.Delegate` or `System.MulticastDelegate`. The compiler generates a type safe `Invoke` method for any delegate type declared. In this example, that means `Invoke` takes a single `string` argument, and has a void return type.

Summary of Practices

You've seen the beginnings of a log component that could be expanded with other writers, and other features. By using delegates in the design, these different components are loosely coupled. This provides several advantages. It's easy to create new output mechanisms and attach them to the system. These other mechanisms only need one method: the method that writes the log message. It's a design that's resilient when new features are added. The contract required for any writer is to implement one method. That method could be a static or instance method. It could be public, private, or any other legal access.

The `Logger` class can make any number of enhancements or changes without introducing breaking changes. Like any class, you cannot modify the public API without the risk of breaking changes. But, because the coupling between the logger and any output engines is only through the delegate, no other types (like interfaces or base classes) are involved. The coupling is as small as possible.

[Next](#)

Introduction to events

12/28/2021 • 3 minutes to read • [Edit Online](#)

[Previous](#)

Events are, like delegates, a *late binding* mechanism. In fact, events are built on the language support for delegates.

Events are a way for an object to broadcast (to all interested components in the system) that something has happened. Any other component can subscribe to the event, and be notified when an event is raised.

You've probably used events in some of your programming. Many graphical systems have an event model to report user interaction. These events would report mouse movement, button presses and similar interactions. That's one of the most common, but certainly not the only scenario where events are used.

You can define events that should be raised for your classes. One important consideration when working with events is that there may not be any object registered for a particular event. You must write your code so that it does not raise events when no listeners are configured.

Subscribing to an event also creates a coupling between two objects (the event source, and the event sink). You need to ensure that the event sink unsubscribes from the event source when no longer interested in events.

Design goals for event support

The language design for events targets these goals:

- Enable very minimal coupling between an event source and an event sink. These two components may not be written by the same organization, and may even be updated on totally different schedules.
- It should be very simple to subscribe to an event, and to unsubscribe from that same event.
- Event sources should support multiple event subscribers. It should also support having no event subscribers attached.

You can see that the goals for events are very similar to the goals for delegates. That's why the event language support is built on the delegate language support.

Language support for events

The syntax for defining events, and subscribing or unsubscribing from events is an extension of the syntax for delegates.

To define an event you use the `event` keyword:

```
public event EventHandler<FileListArgs> Progress;
```

The type of the event (`EventHandler<FileListArgs>` in this example) must be a delegate type. There are a number of conventions that you should follow when declaring an event. Typically, the event delegate type has a void return. Event declarations should be a verb, or a verb phrase. Use past tense when the event reports something that has happened. Use a present tense verb (for example, `Closing`) to report something that is about to happen. Often, using present tense indicates that your class supports some kind of customization behavior. One of the most common scenarios is to support cancellation. For example, a `Closing` event may include an argument that would indicate if the close operation should continue, or not. Other scenarios may enable callers

to modify behavior by updating properties of the event arguments. You may raise an event to indicate a proposed next action an algorithm will take. The event handler may mandate a different action by modifying properties of the event argument.

When you want to raise the event, you call the event handlers using the delegate invocation syntax:

```
Progress?.Invoke(this, new FileListArgs(file));
```

As discussed in the section on [delegates](#), the `?.` operator makes it easy to ensure that you do not attempt to raise the event when there are no subscribers to that event.

You subscribe to an event by using the `+=` operator:

```
EventHandler<FileListArgs> onProgress = (sender, eventArgs) =>  
    Console.WriteLine(eventArgs.FoundFile);  
  
fileLister.Progress += onProgress;
```

The handler method typically has the prefix 'On' followed by the event name, as shown above.

You unsubscribe using the `-=` operator:

```
fileLister.Progress -= onProgress;
```

It's important that you declare a local variable for the expression that represents the event handler. That ensures the unsubscribe removes the handler. If, instead, you used the body of the lambda expression, you are attempting to remove a handler that has never been attached, which does nothing.

In the next article, you'll learn more about typical event patterns, and different variations on this example.

[Next](#)

Standard .NET event patterns

12/28/2021 • 8 minutes to read • [Edit Online](#)

[Previous](#)

.NET events generally follow a few known patterns. Standardizing on these patterns means that developers can leverage knowledge of those standard patterns, which can be applied to any .NET event program.

Let's go through these standard patterns so you will have all the knowledge you need to create standard event sources, and subscribe and process standard events in your code.

Event Delegate Signatures

The standard signature for a .NET event delegate is:

```
void OnEventRaised(object sender, EventArgs args);
```

The return type is void. Events are based on delegates and are multicast delegates. That supports multiple subscribers for any event source. The single return value from a method doesn't scale to multiple event subscribers. Which return value does the event source see after raising an event? Later in this article you'll see how to create event protocols that support event subscribers that report information to the event source.

The argument list contains two arguments: the sender, and the event arguments. The compile time type of `sender` is `System.Object`, even though you likely know a more derived type that would always be correct. By convention, use `object`.

The second argument has typically been a type that is derived from `System.EventArgs`. (You'll see in the [next section](#) that this convention is no longer enforced.) If your event type does not need any additional arguments, you will still provide both arguments. There is a special value, `EventArgs.Empty` that you should use to denote that your event does not contain any additional information.

Let's build a class that lists files in a directory, or any of its subdirectories that follow a pattern. This component raises an event for each file found that matches the pattern.

Using an event model provides some design advantages. You can create multiple event listeners that perform different actions when a sought file is found. Combining the different listeners can create more robust algorithms.

Here is the initial event argument declaration for finding a sought file:

```
public class FileFoundArgs : EventArgs
{
    public string FoundFile { get; }

    public FileFoundArgs(string fileName)
    {
        FoundFile = fileName;
    }
}
```

Even though this type looks like a small, data-only type, you should follow the convention and make it a reference (`class`) type. That means the argument object will be passed by reference, and any updates to the data will be viewed by all subscribers. The first version is an immutable object. You should prefer to make the

properties in your event argument type immutable. That way, one subscriber cannot change the values before another subscriber sees them. (There are exceptions to this, as you'll see below.)

Next, we need to create the event declaration in the `FileSearcher` class. Leveraging the `EventHandler<T>` type means that you don't need to create yet another type definition. You simply use a generic specialization.

Let's fill out the `FileSearcher` class to search for files that match a pattern, and raise the correct event when a match is discovered.

```
public class FileSearcher
{
    public event EventHandler<FileFoundArgs> FileFound;

    public void Search(string directory, string searchPattern)
    {
        foreach (var file in Directory.EnumerateFiles(directory, searchPattern))
        {
            FileFound?.Invoke(this, new FileFoundArgs(file));
        }
    }
}
```

Defining and Raising Field-Like Events

The simplest way to add an event to your class is to declare that event as a public field, as in the preceding example:

```
public event EventHandler<FileFoundArgs> FileFound;
```

This looks like it's declaring a public field, which would appear to be bad object-oriented practice. You want to protect data access through properties, or methods. While this may look like a bad practice, the code generated by the compiler does create wrappers so that the event objects can only be accessed in safe ways. The only operations available on a field-like event are add handler:

```
EventHandler<FileFoundArgs> onFileFound = (sender, eventArgs) =>
{
    Console.WriteLine(eventArgs.FoundFile);
    filesFound++;
};

fileLister.FileFound += onFileFound;
```

and remove handler:

```
fileLister.FileFound -= onFileFound;
```

Note that there's a local variable for the handler. If you used the body of the lambda, the remove would not work correctly. It would be a different instance of the delegate, and silently do nothing.

Code outside the class cannot raise the event, nor can it perform any other operations.

Returning Values from Event Subscribers

Your simple version is working fine. Let's add another feature: Cancellation.

When you raise the found event, listeners should be able to stop further processing, if this file is that last one

sought.

The event handlers do not return a value, so you need to communicate that in another way. The standard event pattern uses the EventArgs object to include fields that event subscribers can use to communicate cancel.

There are two different patterns that could be used, based on the semantics of the Cancel contract. In both cases, you'll add a boolean field to the EventArgs for the found file event.

One pattern would allow any one subscriber to cancel the operation. For this pattern, the new field is initialized to `false`. Any subscriber can change it to `true`. After all subscribers have seen the event raised, the FileSearcher component examines the boolean value and takes action.

The second pattern would only cancel the operation if all subscribers wanted the operation cancelled. In this pattern, the new field is initialized to indicate the operation should cancel, and any subscriber could change it to indicate the operation should continue. After all subscribers have seen the event raised, the FileSearcher component examines the boolean and takes action. There is one extra step in this pattern: the component needs to know if any subscribers have seen the event. If there are no subscribers, the field would indicate a cancel incorrectly.

Let's implement the first version for this sample. You need to add a boolean field named `CancelRequested` to the `FileFoundArgs` type:

```
public class FileFoundArgs : EventArgs
{
    public string FoundFile { get; }
    public bool CancelRequested { get; set; }

    public FileFoundArgs(string fileName)
    {
        FoundFile = fileName;
    }
}
```

This new Field is automatically initialized to `false`, the default value for a Boolean field, so you don't cancel accidentally. The only other change to the component is to check the flag after raising the event to see if any of the subscribers have requested a cancellation:

```
public void List(string directory, string searchPattern)
{
    foreach (var file in Directory.EnumerateFiles(directory, searchPattern))
    {
        var args = new FileFoundArgs(file);
        FileFound?.Invoke(this, args);
        if (args.CancelRequested)
            break;
    }
}
```

One advantage of this pattern is that it isn't a breaking change. None of the subscribers requested a cancellation before, and they still are not. None of the subscriber code needs updating unless they want to support the new cancel protocol. It's very loosely coupled.

Let's update the subscriber so that it requests a cancellation once it finds the first executable:

```

EventHandler<FileFoundArgs> onFileFound = (sender, eventArgs) =>
{
    Console.WriteLine(eventArgs.FoundFile);
    eventArgs.CancelRequested = true;
};

```

Adding Another Event Declaration

Let's add one more feature, and demonstrate other language idioms for events. Let's add an overload of the `Search` method that traverses all subdirectories in search of files.

This could get to be a lengthy operation in a directory with many sub-directories. Let's add an event that gets raised when each new directory search begins. This enables subscribers to track progress, and update the user as to progress. All the samples you've created so far are public. Let's make this one an internal event. That means you can also make the types used for the arguments internal as well.

You'll start by creating the new EventArgs derived class for reporting the new directory and progress.

```

internal class SearchDirectoryArgs : EventArgs
{
    internal string CurrentSearchDirectory { get; }
    internal int TotalDirs { get; }
    internal int CompletedDirs { get; }

    internal SearchDirectoryArgs(string dir, int totalDirs, int completedDirs)
    {
        CurrentSearchDirectory = dir;
        TotalDirs = totalDirs;
        CompletedDirs = completedDirs;
    }
}

```

Again, you can follow the recommendations to make an immutable reference type for the event arguments.

Next, define the event. This time, you'll use a different syntax. In addition to using the field syntax, you can explicitly create the property, with add and remove handlers. In this sample, you won't need extra code in those handlers, but this shows how you would create them.

```

internal event EventHandler<SearchDirectoryArgs> DirectoryChanged
{
    add { directoryChanged += value; }
    remove { directoryChanged -= value; }
}
private EventHandler<SearchDirectoryArgs> directoryChanged;

```

In many ways, the code you write here mirrors the code the compiler generates for the field event definitions you've seen earlier. You create the event using syntax very similar to that used for [properties](#). Notice that the handlers have different names: `add` and `remove`. These are called to subscribe to the event, or unsubscribe from the event. Notice that you also must declare a private backing field to store the event variable. It is initialized to null.

Next, let's add the overload of the `Search` method that traverses subdirectories and raises both events. The easiest way to accomplish this is to use a default argument to specify that you want to search all directories:


```

public void Search(string directory, string searchPattern, bool searchSubDirs = false)
{
    if (searchSubDirs)
    {
        var allDirectories = Directory.GetDirectories(directory, " *.*", SearchOption.AllDirectories);
        var completedDirs = 0;
        var totalDirs = allDirectories.Length + 1;
        foreach (var dir in allDirectories)
        {
            directoryChanged?.Invoke(this,
                new SearchDirectoryArgs(dir, totalDirs, completedDirs++));
            // Search 'dir' and its subdirectories for files that match the search pattern:
            SearchDirectory(dir, searchPattern);
        }
        // Include the Current Directory:
        directoryChanged?.Invoke(this,
            new SearchDirectoryArgs(directory, totalDirs, completedDirs++));
        SearchDirectory(directory, searchPattern);
    }
    else
    {
        SearchDirectory(directory, searchPattern);
    }
}

private void SearchDirectory(string directory, string searchPattern)
{
    foreach (var file in Directory.EnumerateFiles(directory, searchPattern))
    {
        var args = new FileFoundArgs(file);
        FileFound?.Invoke(this, args);
        if (args.CancelRequested)
            break;
    }
}

```

At this point, you can run the application calling the overload for searching all sub-directories. There are no subscribers on the new `DirectoryChanged` event, but using the `?.Invoke()` idiom ensures that this works correctly.

Let's add a handler to write a line that shows the progress in the console window.

```

fileLister.DirectoryChanged += (sender, eventArgs) =>
{
    Console.WriteLine($"Entering '{eventArgs.CurrentSearchDirectory}'");
    Console.WriteLine($" {eventArgs.CompletedDirs} of {eventArgs.TotalDirs} completed...");
};

```

You've seen patterns that are followed throughout the .NET ecosystem. By learning these patterns and conventions, you'll be writing idiomatic C# and .NET quickly.

Next, you'll see some changes in these patterns in the most recent release of .NET.

[Next](#)

The Updated .NET Core Event Pattern

12/28/2021 • 3 minutes to read • [Edit Online](#)

[Previous](#)

The previous article discussed the most common event patterns. .NET Core has a more relaxed pattern. In this version, the `EventHandler<TEventArgs>` definition no longer has the constraint that `TEventArgs` must be a class derived from `System.EventArgs`.

This increases flexibility for you, and is backwards compatible. Let's start with the flexibility. The class `System.EventArgs` introduces one method: `MemberwiseClone()`, which creates a shallow copy of the object. That method must use reflection in order to implement its functionality for any class derived from `EventArgs`. That functionality is easier to create in a specific derived class. That effectively means that deriving from `System.EventArgs` is a constraint that limits your designs, but does not provide any additional benefit. In fact, you can change the definitions of `FileFoundArgs` and `SearchDirectoryArgs` so that they do not derive from `EventArgs`. The program will work exactly the same.

You could also change the `SearchDirectoryArgs` to a struct, if you make one more change:

```
internal struct SearchDirectoryArgs
{
    internal string CurrentSearchDirectory { get; }
    internal int TotalDirs { get; }
    internal int CompletedDirs { get; }

    internal SearchDirectoryArgs(string dir, int totalDirs, int completedDirs) : this()
    {
        CurrentSearchDirectory = dir;
        TotalDirs = totalDirs;
        CompletedDirs = completedDirs;
    }
}
```

The additional change is to call the parameterless constructor before entering the constructor that initializes all the fields. Without that addition, the rules of C# would report that the properties are being accessed before they have been assigned.

You should not change the `FileFoundArgs` from a class (reference type) to a struct (value type). That's because the protocol for handling cancel requires that the event arguments are passed by reference. If you made the same change, the file search class could never observe any changes made by any of the event subscribers. A new copy of the structure would be used for each subscriber, and that copy would be a different copy than the one seen by the file search object.

Next, let's consider how this change can be backwards compatible. The removal of the constraint does not affect any existing code. Any existing event argument types do still derive from `System.EventArgs`. Backwards compatibility is one major reason why they will continue to derive from `System.EventArgs`. Any existing event subscribers will be subscribers to an event that followed the classic pattern.

Following similar logic, any event argument type created now would not have any subscribers in any existing codebases. New event types that do not derive from `System.EventArgs` will not break those codebases.

Events with Async subscribers

You have one final pattern to learn: How to correctly write event subscribers that call async code. The challenge is described in the article on [async and await](#). Async methods can have a void return type, but that is strongly discouraged. When your event subscriber code calls an async method, you have no choice but to create an `async void` method. The event handler signature requires it.

You need to reconcile this opposing guidance. Somehow, you must create a safe `async void` method. The basics of the pattern you need to implement are below:

```
worker.StartWorking += async (sender, eventArgs) =>
{
    try
    {
        await DoWorkAsync();
    }
    catch (Exception e)
    {
        //Some form of logging.
        Console.WriteLine($"Async task failure: {e.ToString()}");
        // Consider gracefully, and quickly exiting.
    }
};
```

First, notice that the handler is marked as an async handler. Because it is being assigned to an event handler delegate type, it will have a void return type. That means you must follow the pattern shown in the handler, and not allow any exceptions to be thrown out of the context of the async handler. Because it does not return a task, there is no task that can report the error by entering the faulted state. Because the method is async, the method can't simply throw the exception. (The calling method has continued execution because it is `async`.) The actual runtime behavior will be defined differently for different environments. It may terminate the thread or the process that owns the thread, or leave the process in an indeterminate state. All of these potential outcomes are highly undesirable.

That's why you should wrap the await statement for the async Task in your own try block. If it does cause a faulted task, you can log the error. If it is an error from which your application cannot recover, you can exit the program quickly and gracefully

Those are the major updates to the .NET event pattern. You will see many examples of the earlier versions in the libraries you work with. However, you should understand what the latest patterns are as well.

The next article in this series helps you distinguish between using `delegates` and `events` in your designs. They are similar concepts, and that article will help you make the best decision for your programs.

[Next](#)

Distinguishing Delegates and Events

12/28/2021 • 3 minutes to read • [Edit Online](#)

[Previous](#)

Developers that are new to the .NET Core platform often struggle when deciding between a design based on `delegates` and a design based on `events`. The choice of delegates or events is often difficult, because the two language features are similar. Events are even built using the language support for delegates.

They both offer a late binding scenario: they enable scenarios where a component communicates by calling a method that is only known at run time. They both support single and multiple subscriber methods. You may find this referred to as singlecast and multicast support. They both support similar syntax for adding and removing handlers. Finally, raising an event and calling a delegate use exactly the same method call syntax. They even both support the same `Invoke()` method syntax for use with the `?.` operator.

With all those similarities, it is easy to have trouble determining when to use which.

Listening to Events is Optional

The most important consideration in determining which language feature to use is whether or not there must be an attached subscriber. If your code must call the code supplied by the subscriber, you should use a design based on delegates when you need to implement callback. If your code can complete all its work without calling any subscribers, you should use a design based on events.

Consider the examples built during this section. The code you built using `List.Sort()` must be given a comparer function in order to properly sort the elements. LINQ queries must be supplied with delegates in order to determine what elements to return. Both used a design built with delegates.

Consider the `Progress` event. It reports progress on a task. The task continues to proceed whether or not there are any listeners. The `FileSearcher` is another example. It would still search and find all the files that were sought, even with no event subscribers attached. UX controls still work correctly, even when there are no subscribers listening to the events. They both use designs based on events.

Return Values Require Delegates

Another consideration is the method prototype you would want for your delegate method. As you've seen, the delegates used for events all have a void return type. You've also seen that there are idioms to create event handlers that do pass information back to event sources through modifying properties of the event argument object. While these idioms do work, they are not as natural as returning a value from a method.

Notice that these two heuristics may often both be present: If your delegate method returns a value, it will likely impact the algorithm in some way.

Events Have Private Invocation

Classes other than the one in which an event is contained can only add and remove event listeners; only the class containing the event can invoke the event. Events are typically public class members. By comparison, delegates are often passed as parameters and stored as private class members, if they are stored at all.

Event Listeners Often Have Longer Lifetimes

That event listeners have longer lifetimes is a slightly weaker justification. However, you may find that event-

based designs are more natural when the event source will be raising events over a long period of time. You can see examples of event-based design for UX controls on many systems. Once you subscribe to an event, the event source may raise events throughout the lifetime of the program. (You can unsubscribe from events when you no longer need them.)

Contrast that with many delegate-based designs, where a delegate is used as an argument to a method, and the delegate is not used after that method returns.

Evaluate Carefully

The above considerations are not hard and fast rules. Instead, they represent guidance that can help you decide which choice is best for your particular usage. Because they are similar, you can even prototype both, and consider which would be more natural to work with. They both handle late binding scenarios well. Use the one that communicates your design the best.

Language Integrated Query (LINQ)

12/28/2021 • 3 minutes to read • [Edit Online](#)

Language-Integrated Query (LINQ) is the name for a set of technologies based on the integration of query capabilities directly into the C# language. Traditionally, queries against data are expressed as simple strings without type checking at compile time or IntelliSense support. Furthermore, you have to learn a different query language for each type of data source: SQL databases, XML documents, various Web services, and so on. With LINQ, a query is a first-class language construct, just like classes, methods, events.

For a developer who writes queries, the most visible "language-integrated" part of LINQ is the query expression. Query expressions are written in a declarative *query syntax*. By using query syntax, you can perform filtering, ordering, and grouping operations on data sources with a minimum of code. You use the same basic query expression patterns to query and transform data in SQL databases, ADO .NET Datasets, XML documents and streams, and .NET collections.

The following example shows the complete query operation. The complete operation includes creating a data source, defining the query expression, and executing the query in a `foreach` statement.

```
class LINQQueryExpressions
{
    static void Main()
    {
        // Specify the data source.
        int[] scores = new int[] { 97, 92, 81, 60 };

        // Define the query expression.
        IEnumerable<int> scoreQuery =
            from score in scores
            where score > 80
            select score;

        // Execute the query.
        foreach (int i in scoreQuery)
        {
            Console.Write(i + " ");
        }
    }
}
// Output: 97 92 81
```

Query expression overview

- Query expressions can be used to query and to transform data from any LINQ-enabled data source. For example, a single query can retrieve data from a SQL database, and produce an XML stream as output.
- Query expressions are easy to grasp because they use many familiar C# language constructs.
- The variables in a query expression are all strongly typed, although in many cases you do not have to provide the type explicitly because the compiler can infer it. For more information, see [Type relationships in LINQ query operations](#).
- A query is not executed until you iterate over the query variable, for example, in a `foreach` statement. For more information, see [Introduction to LINQ queries](#).

- At compile time, query expressions are converted to Standard Query Operator method calls according to the rules set forth in the C# specification. Any query that can be expressed by using query syntax can also be expressed by using method syntax. However, in most cases query syntax is more readable and concise. For more information, see [C# language specification](#) and [Standard query operators overview](#).
- As a rule when you write LINQ queries, we recommend that you use query syntax whenever possible and method syntax whenever necessary. There is no semantic or performance difference between the two different forms. Query expressions are often more readable than equivalent expressions written in method syntax.
- Some query operations, such as [Count](#) or [Max](#), have no equivalent query expression clause and must therefore be expressed as a method call. Method syntax can be combined with query syntax in various ways. For more information, see [Query syntax and method syntax in LINQ](#).
- Query expressions can be compiled to expression trees or to delegates, depending on the type that the query is applied to. [IEnumerable<T>](#) queries are compiled to delegates. [IQueryable](#) and [IQueryable<T>](#) queries are compiled to expression trees. For more information, see [Expression trees](#).

Next steps

To learn more details about LINQ, start by becoming familiar with some basic concepts in [Query expression basics](#), and then read the documentation for the LINQ technology in which you are interested:

- XML documents: [LINQ to XML](#)
- ADO.NET Entity Framework: [LINQ to entities](#)
- .NET collections, files, strings and so on: [LINQ to objects](#)

To gain a deeper understanding of LINQ in general, see [LINQ in C#](#).

To start working with LINQ in C#, see the tutorial [Working with LINQ](#).

Query expression basics

12/28/2021 • 12 minutes to read • [Edit Online](#)

This article introduces the basic concepts related to query expressions in C#.

What is a query and what does it do?

A *query* is a set of instructions that describes what data to retrieve from a given data source (or sources) and what shape and organization the returned data should have. A query is distinct from the results that it produces.

Generally, the source data is organized logically as a sequence of elements of the same kind. For example, a SQL database table contains a sequence of rows. In an XML file, there is a "sequence" of XML elements (although these are organized hierarchically in a tree structure). An in-memory collection contains a sequence of objects.

From an application's viewpoint, the specific type and structure of the original source data is not important. The application always sees the source data as an `IEnumerable<T>` or `IQueryable<T>` collection. For example, in LINQ to XML, the source data is made visible as an `IEnumerable<XElement>`.

Given this source sequence, a query may do one of three things:

- Retrieve a subset of the elements to produce a new sequence without modifying the individual elements. The query may then sort or group the returned sequence in various ways, as shown in the following example (assume `scores` is an `int[]`):

```
IEnumerable<int> highScoresQuery =  
    from score in scores  
    where score > 80  
    orderby score descending  
    select score;
```

- Retrieve a sequence of elements as in the previous example but transform them to a new type of object. For example, a query may retrieve only the last names from certain customer records in a data source. Or it may retrieve the complete record and then use it to construct another in-memory object type or even XML data before generating the final result sequence. The following example shows a projection from an `int` to a `string`. Note the new type of `highScoresQuery`.

```
IEnumerable<string> highScoresQuery2 =  
    from score in scores  
    where score > 80  
    orderby score descending  
    select $"The score is {score}";
```

- Retrieve a singleton value about the source data, such as:
 - The number of elements that match a certain condition.
 - The element that has the greatest or least value.
 - The first element that matches a condition, or the sum of particular values in a specified set of elements. For example, the following query returns the number of scores greater than 80 from the `scores` integer array:


```
int highScoreCount =  
    (from score in scores  
     where score > 80  
     select score)  
    .Count();
```

In the previous example, note the use of parentheses around the query expression before the call to the `Count` method. You can also express this by using a new variable to store the concrete result. This technique is more readable because it keeps the variable that stores the query separate from the query that stores a result.

```
IEnumerable<int> highScoresQuery3 =  
    from score in scores  
    where score > 80  
    select score;  
  
int scoreCount = highScoresQuery3.Count();
```

In the previous example, the query is executed in the call to `Count`, because `Count` must iterate over the results in order to determine the number of elements returned by `highScoresQuery`.

What is a query expression?

A *query expression* is a query expressed in query syntax. A query expression is a first-class language construct. It is just like any other expression and can be used in any context in which a C# expression is valid. A query expression consists of a set of clauses written in a declarative syntax similar to SQL or XQuery. Each clause in turn contains one or more C# expressions, and these expressions may themselves be either a query expression or contain a query expression.

A query expression must begin with a `from` clause and must end with a `select` or `group` clause. Between the first `from` clause and the last `select` or `group` clause, it can contain one or more of these optional clauses: `where`, `orderby`, `join`, `let` and even additional `from` clauses. You can also use the `into` keyword to enable the result of a `join` or `group` clause to serve as the source for additional query clauses in the same query expression.

Query variable

In LINQ, a query variable is any variable that stores a *query* instead of the *results* of a query. More specifically, a query variable is always an enumerable type that will produce a sequence of elements when it is iterated over in a `foreach` statement or a direct call to its `IEnumerator.MoveNext` method.

The following code example shows a simple query expression with one data source, one filtering clause, one ordering clause, and no transformation of the source elements. The `select` clause ends the query.

```

static void Main()
{
    // Data source.
    int[] scores = { 90, 71, 82, 93, 75, 82 };

    // Query Expression.
    IEnumerable<int> scoreQuery = //query variable
        from score in scores //required
        where score > 80 // optional
        orderby score descending // optional
        select score; //must end with select or group

    // Execute the query to produce the results
    foreach (int testScore in scoreQuery)
    {
        Console.WriteLine(testScore);
    }
}
// Outputs: 93 90 82 82

```

In the previous example, `scoreQuery` is a *query variable*, which is sometimes referred to as just a *query*. The query variable stores no actual result data, which is produced in the `foreach` loop. And when the `foreach` statement executes, the query results are not returned through the query variable `scoreQuery`. Rather, they are returned through the iteration variable `testScore`. The `scoreQuery` variable can be iterated in a second `foreach` loop. It will produce the same results as long as neither it nor the data source has been modified.

A query variable may store a query that is expressed in query syntax or method syntax, or a combination of the two. In the following examples, both `queryMajorCities` and `queryMajorCities2` are query variables:

```

//Query syntax
IEnumerable<City> queryMajorCities =
    from city in cities
    where city.Population > 100000
    select city;

// Method-based syntax
IEnumerable<City> queryMajorCities2 = cities.Where(c => c.Population > 100000);

```

On the other hand, the following two examples show variables that are not query variables even though each is initialized with a query. They are not query variables because they store results:

```

int highestScore =
    (from score in scores
     select score)
     .Max();

// or split the expression
IEnumerable<int> scoreQuery =
    from score in scores
    select score;

int highScore = scoreQuery.Max();
// the following returns the same result
int highScore = scores.Max();

List<City> largeCitiesList =
    (from country in countries
     from city in country.Cities
     where city.Population > 10000
     select city)
     .ToList();

// or split the expression
IEnumerable<City> largeCitiesQuery =
    from country in countries
    from city in country.Cities
    where city.Population > 10000
    select city;

List<City> largeCitiesList2 = largeCitiesQuery.ToList();

```

For more information about the different ways to express queries, see [Query syntax and method syntax in LINQ](#).

Explicit and implicit typing of query variables

This documentation usually provides the explicit type of the query variable in order to show the type relationship between the query variable and the [select clause](#). However, you can also use the `var` keyword to instruct the compiler to infer the type of a query variable (or any other local variable) at compile time. For example, the query example that was shown previously in this topic can also be expressed by using implicit typing:

```

// Use of var is optional here and in all queries.
// queryCities is an IEnumerable<City> just as
// when it is explicitly typed.
var queryCities =
    from city in cities
    where city.Population > 100000
    select city;

```

For more information, see [Implicitly typed local variables](#) and [Type relationships in LINQ query operations](#).

Starting a query expression

A query expression must begin with a `from` clause. It specifies a data source together with a range variable. The range variable represents each successive element in the source sequence as the source sequence is being traversed. The range variable is strongly typed based on the type of elements in the data source. In the following example, because `countries` is an array of `Country` objects, the range variable is also typed as `Country`. Because the range variable is strongly typed, you can use the dot operator to access any available members of the type.

```
IEnumerable<Country> countryAreaQuery =  
    from country in countries  
    where country.Area > 500000 //sq km  
    select country;
```

The range variable is in scope until the query is exited either with a semicolon or with a [continuation](#) clause.

A query expression may contain multiple `from` clauses. Use additional `from` clauses when each element in the source sequence is itself a collection or contains a collection. For example, assume that you have a collection of `Country` objects, each of which contains a collection of `City` objects named `Cities`. To query the `City` objects in each `Country`, use two `from` clauses as shown here:

```
IEnumerable<City> cityQuery =  
    from country in countries  
    from city in country.Cities  
    where city.Population > 10000  
    select city;
```

For more information, see [from clause](#).

Ending a query expression

A query expression must end with either a `group` clause or a `select` clause.

group clause

Use the `group` clause to produce a sequence of groups organized by a key that you specify. The key can be any data type. For example, the following query creates a sequence of groups that contains one or more `Country` objects and whose key is a `char` type with value being the first letter of countries' names.

```
var queryCountryGroups =  
    from country in countries  
    group country by country.Name[0];
```

For more information about grouping, see [group clause](#).

select clause

Use the `select` clause to produce all other types of sequences. A simple `select` clause just produces a sequence of the same type of objects as the objects that are contained in the data source. In this example, the data source contains `Country` objects. The `orderby` clause just sorts the elements into a new order and the `select` clause produces a sequence of the reordered `Country` objects.

```
IEnumerable<Country> sortedQuery =  
    from country in countries  
    orderby country.Area  
    select country;
```

The `select` clause can be used to transform source data into sequences of new types. This transformation is also named a *projection*. In the following example, the `select` clause *projects* a sequence of anonymous types which contains only a subset of the fields in the original element. Note that the new objects are initialized by using an object initializer.

```
// Here var is required because the query
// produces an anonymous type.
var queryNameAndPop =
    from country in countries
    select new { Name = country.Name, Pop = country.Population };
```

For more information about all the ways that a `select` clause can be used to transform source data, see [select clause](#).

Continuations with `into`

You can use the `into` keyword in a `select` or `group` clause to create a temporary identifier that stores a query. Do this when you must perform additional query operations on a query after a grouping or select operation. In the following example `countries` are grouped according to population in ranges of 10 million. After these groups are created, additional clauses filter out some groups, and then to sort the groups in ascending order. To perform those additional operations, the continuation represented by `countryGroup` is required.

```
// percentileQuery is an IEnumerable<IGrouping<int, Country>>
var percentileQuery =
    from country in countries
    let percentile = (int) country.Population / 10_000_000
    group country by percentile into countryGroup
    where countryGroup.Key >= 20
    orderby countryGroup.Key
    select countryGroup;

// grouping is an IGrouping<int, Country>
foreach (var grouping in percentileQuery)
{
    Console.WriteLine(grouping.Key);
    foreach (var country in grouping)
        Console.WriteLine(country.Name + ":" + country.Population);
}
```

For more information, see [into](#).

Filtering, ordering, and joining

Between the starting `from` clause, and the ending `select` or `group` clause, all other clauses (`where`, `join`, `orderby`, `from`, `let`) are optional. Any of the optional clauses may be used zero times or multiple times in a query body.

where clause

Use the `where` clause to filter out elements from the source data based on one or more predicate expressions. The `where` clause in the following example has one predicate with two conditions.

```
IEnumerable<City> queryCityPop =
    from city in cities
    where city.Population < 200000 && city.Population > 100000
    select city;
```

For more information, see [where clause](#).

orderby clause

Use the `orderby` clause to sort the results in either ascending or descending order. You can also specify secondary sort orders. The following example performs a primary sort on the `country` objects by using the `Area` property. It then performs a secondary sort by using the `Population` property.

```

IEnumerable<Country> querySortedCountries =
    from country in countries
    orderby country.Area, country.Population descending
    select country;

```

The `ascending` keyword is optional; it is the default sort order if no order is specified. For more information, see [orderby clause](#).

join clause

Use the `join` clause to associate and/or combine elements from one data source with elements from another data source based on an equality comparison between specified keys in each element. In LINQ, join operations are performed on sequences of objects whose elements are different types. After you have joined two sequences, you must use a `select` or `group` statement to specify which element to store in the output sequence. You can also use an anonymous type to combine properties from each set of associated elements into a new type for the output sequence. The following example associates `prod` objects whose `Category` property matches one of the categories in the `categories` string array. Products whose `Category` does not match any string in `categories` are filtered out. The `select` statement projects a new type whose properties are taken from both `cat` and `prod`.

```

var categoryQuery =
    from cat in categories
    join prod in products on cat equals prod.Category
    select new { Category = cat, Name = prod.Name };

```

You can also perform a group join by storing the results of the `join` operation into a temporary variable by using the `into` keyword. For more information, see [join clause](#).

let clause

Use the `let` clause to store the result of an expression, such as a method call, in a new range variable. In the following example, the range variable `firstName` stores the first element of the array of strings that is returned by `Split`.

```

string[] names = { "Svetlana Omelchenko", "Claire O'Donnell", "Sven Mortensen", "Cesar Garcia" };
IEnumerable<string> queryFirstNames =
    from name in names
    let firstName = name.Split(' ')[0]
    select firstName;

foreach (string s in queryFirstNames)
    Console.Write(s + " ");
//Output: Svetlana Claire Sven Cesar

```

For more information, see [let clause](#).

Subqueries in a query expression

A query clause may itself contain a query expression, which is sometimes referred to as a *subquery*. Each subquery starts with its own `from` clause that does not necessarily point to the same data source in the first `from` clause. For example, the following query shows a query expression that is used in the select statement to retrieve the results of a grouping operation.

```
var queryGroupMax =  
    from student in students  
    group student by student.GradeLevel into studentGroup  
    select new  
    {  
        Level = studentGroup.Key,  
        HighestScore =  
            (from student2 in studentGroup  
             select student2.Scores.Average())  
            .Max()  
    };
```

For more information, see [Perform a subquery on a grouping operation](#).

See also

- [C# programming guide](#)
- [Language Integrated Query \(LINQ\)](#)
- [Query keywords \(LINQ\)](#)
- [Standard query operators overview](#)

LINQ in C#

12/28/2021 • 2 minutes to read • [Edit Online](#)

This section contains links to topics that provide more detailed information about LINQ.

In this section

[Introduction to LINQ queries](#)

Describes the three parts of the basic LINQ query operation that are common across all languages and data sources.

[LINQ and generic types](#)

Provides a brief introduction to generic types as they are used in LINQ.

[Data transformations with LINQ](#)

Describes the various ways that you can transform data retrieved in queries.

[Type relationships in LINQ query operations](#)

Describes how types are preserved and/or transformed in the three parts of a LINQ query operation

[Query syntax and method syntax in LINQ](#)

Compares method syntax and query syntax as two ways to express a LINQ query.

[C# features that support LINQ](#)

Describes the language constructs in C# that support LINQ.

Related sections

[LINQ query expressions](#)

Includes an overview of queries in LINQ and provides links to additional resources.

[Standard query operators overview](#)

Introduces the standard methods used in LINQ.

Write LINQ queries in C#

12/28/2021 • 4 minutes to read • [Edit Online](#)

This article shows the three ways in which you can write a LINQ query in C#:

1. Use query syntax.
2. Use method syntax.
3. Use a combination of query syntax and method syntax.

The following examples demonstrate some simple LINQ queries by using each approach listed previously. In general, the rule is to use (1) whenever possible, and use (2) and (3) whenever necessary.

NOTE

These queries operate on simple in-memory collections; however, the basic syntax is identical to that used in LINQ to Entities and LINQ to XML.

Example - Query syntax

The recommended way to write most queries is to use *query syntax* to create *query expressions*. The following example shows three query expressions. The first query expression demonstrates how to filter or restrict results by applying conditions with a `where` clause. It returns all elements in the source sequence whose values are greater than 7 or less than 3. The second expression demonstrates how to order the returned results. The third expression demonstrates how to group results according to a key. This query returns two groups based on the first letter of the word.

```
// Query #1.
List<int> numbers = new List<int>() { 5, 4, 1, 3, 9, 8, 6, 7, 2, 0 };

// The query variable can also be implicitly typed by using var
IEnumerable<int> filteringQuery =
    from num in numbers
    where num < 3 || num > 7
    select num;

// Query #2.
IEnumerable<int> orderingQuery =
    from num in numbers
    where num < 3 || num > 7
    orderby num ascending
    select num;

// Query #3.
string[] groupingQuery = { "carrots", "cabbage", "broccoli", "beans", "barley" };
IEnumerable<IGrouping<char, string>> queryFoodGroups =
    from item in groupingQuery
    group item by item[0];
```

Note that the type of the queries is `IEnumerable<T>`. All of these queries could be written using `var` as shown in the following example:

```
var query = from num in numbers...
```

In each previous example, the queries do not actually execute until you iterate over the query variable in a `foreach` statement or other statement. For more information, see [Introduction to LINQ Queries](#).

Example - Method syntax

Some query operations must be expressed as a method call. The most common such methods are those that return singleton numeric values, such as [Sum](#), [Max](#), [Min](#), [Average](#), and so on. These methods must always be called last in any query because they represent only a single value and cannot serve as the source for an additional query operation. The following example shows a method call in a query expression:

```
List<int> numbers1 = new List<int>() { 5, 4, 1, 3, 9, 8, 6, 7, 2, 0 };
List<int> numbers2 = new List<int>() { 15, 14, 11, 13, 19, 18, 16, 17, 12, 10 };
// Query #4.
double average = numbers1.Average();

// Query #5.
IEnumerable<int> concatenationQuery = numbers1.Concat(numbers2);
```

If the method has Action or Func parameters, these are provided in the form of a [lambda](#) expression, as shown in the following example:

```
// Query #6.
IEnumerable<int> largeNumbersQuery = numbers2.Where(c => c > 15);
```

In the previous queries, only Query #4 executes immediately. This is because it returns a single value, and not a generic `IEnumerable<T>` collection. The method itself has to use `foreach` in order to compute its value.

Each of the previous queries can be written by using implicit typing with `var`, as shown in the following example:

```
// var is used for convenience in these queries
var average = numbers1.Average();
var concatenationQuery = numbers1.Concat(numbers2);
var largeNumbersQuery = numbers2.Where(c => c > 15);
```

Example - Mixed query and method syntax

This example shows how to use method syntax on the results of a query clause. Just enclose the query expression in parentheses, and then apply the dot operator and call the method. In the following example, query #7 returns a count of the numbers whose value is between 3 and 7. In general, however, it is better to use a second variable to store the result of the method call. In this manner, the query is less likely to be confused with the results of the query.

```
// Query #7.

// Using a query expression with method syntax
int numCount1 =
    (from num in numbers1
     where num < 3 || num > 7
     select num).Count();

// Better: Create a new variable to store
// the method call result
IEnumerable<int> numbersQuery =
    from num in numbers1
    where num < 3 || num > 7
    select num;

int numCount2 = numbersQuery.Count();
```

Because Query #7 returns a single value and not a collection, the query executes immediately.

The previous query can be written by using implicit typing with `var`, as follows:

```
var numCount = (from num in numbers...
```

It can be written in method syntax as follows:

```
var numCount = numbers.Where(n => n < 3 || n > 7).Count();
```

It can be written by using explicit typing, as follows:

```
int numCount = numbers.Where(n => n < 3 || n > 7).Count();
```

See also

- [Walkthrough: Writing Queries in C#](#)
- [Language Integrated Query \(LINQ\)](#)
- [where clause](#)

Query a collection of objects

12/28/2021 • 2 minutes to read • [Edit Online](#)

This example shows how to perform a simple query over a list of `Student` objects. Each `Student` object contains some basic information about the student, and a list that represents the student's scores on four examinations.

This application serves as the framework for many other examples in this section that use the same `students` data source.

Example

The following query returns the students who received a score of 90 or greater on their first exam.

```
public class Student
{
    #region data
    public enum GradeLevel { FirstYear = 1, SecondYear, ThirdYear, FourthYear };

    public string FirstName { get; set; }
    public string LastName { get; set; }
    public int Id { get; set; }
    public GradeLevel Year;
    public List<int> ExamScores;

    protected static List<Student> students = new List<Student>
    {
        new Student {FirstName = "Terry", LastName = "Adams", Id = 120,
            Year = GradeLevel.SecondYear,
            ExamScores = new List<int> { 99, 82, 81, 79}},
        new Student {FirstName = "Fadi", LastName = "Fakhouri", Id = 116,
            Year = GradeLevel.ThirdYear,
            ExamScores = new List<int> { 99, 86, 90, 94}},
        new Student {FirstName = "Hanying", LastName = "Feng", Id = 117,
            Year = GradeLevel.FirstYear,
            ExamScores = new List<int> { 93, 92, 80, 87}},
        new Student {FirstName = "Cesar", LastName = "Garcia", Id = 114,
            Year = GradeLevel.FourthYear,
            ExamScores = new List<int> { 97, 89, 85, 82}},
        new Student {FirstName = "Debra", LastName = "Garcia", Id = 115,
            Year = GradeLevel.ThirdYear,
            ExamScores = new List<int> { 35, 72, 91, 70}},
        new Student {FirstName = "Hugo", LastName = "Garcia", Id = 118,
            Year = GradeLevel.SecondYear,
            ExamScores = new List<int> { 92, 90, 83, 78}},
        new Student {FirstName = "Sven", LastName = "Mortensen", Id = 113,
            Year = GradeLevel.FirstYear,
            ExamScores = new List<int> { 88, 94, 65, 91}},
        new Student {FirstName = "Claire", LastName = "O'Donnell", Id = 112,
            Year = GradeLevel.FourthYear,
            ExamScores = new List<int> { 75, 84, 91, 39}},
        new Student {FirstName = "Svetlana", LastName = "Omelchenko", Id = 111,
            Year = GradeLevel.SecondYear,
            ExamScores = new List<int> { 97, 92, 81, 60}},
        new Student {FirstName = "Lance", LastName = "Tucker", Id = 119,
            Year = GradeLevel.ThirdYear,
            ExamScores = new List<int> { 68, 79, 88, 92}},
        new Student {FirstName = "Michael", LastName = "Tucker", Id = 122,
            Year = GradeLevel.FirstYear,
            ExamScores = new List<int> { 94, 92, 91, 91}},
    }
```

```

        new Student {FirstName = "Eugene", LastName = "Zabokritski", Id = 121,
                      Year = GradeLevel.FourthYear,
                      ExamScores = new List<int> { 96, 85, 91, 60}}
    };
#endregion

// Helper method, used in GroupByRange.
protected static int GetPercentile(Student s)
{
    double avg = s.ExamScores.Average();
    return avg > 0 ? (int)avg / 10 : 0;
}

public static void QueryHighScores(int exam, int score)
{
    var highScores = from student in students
                     where student.ExamScores[exam] > score
                     select new {Name = student.FirstName, Score = student.ExamScores[exam]};

    foreach (var item in highScores)
    {
        Console.WriteLine($"{item.Name,-15}{item.Score}");
    }
}

}

public class Program
{
    public static void Main()
    {
        Student.QueryHighScores(1, 90);

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit");
        Console.ReadKey();
    }
}

```

This query is intentionally simple to enable you to experiment. For example, you can try more conditions in the `where` clause, or use an `orderby` clause to sort the results.

See also

- [Language Integrated Query \(LINQ\)](#)
- [String interpolation](#)

How to return a query from a method (C# Programming Guide)

12/28/2021 • 2 minutes to read • [Edit Online](#)

This example shows how to return a query from a method as the return value and as an `out` parameter.

Query objects are composable, meaning that you can return a query from a method. Objects that represent queries do not store the resulting collection, but rather the steps to produce the results when needed. The advantage of returning query objects from methods is that they can be further composed or modified. Therefore any return value or `out` parameter of a method that returns a query must also have that type. If a method materializes a query into a concrete `List<T>` or `Array` type, it is considered to be returning the query results instead of the query itself. A query variable that is returned from a method can still be composed or modified.

Example

In the following example, the first method returns a query as a return value, and the second method returns a query as an `out` parameter. Note that in both cases it is a query that is returned, not query results.

```
class MQ
{
    // QueryMethod1 returns a query as its value.
    IEnumerable<string> QueryMethod1(ref int[] ints)
    {
        var intsToStrings = from i in ints
                            where i > 4
                            select i.ToString();

        return intsToStrings;
    }

    // QueryMethod2 returns a query as the value of parameter returnQ.
    void QueryMethod2(ref int[] ints, out IEnumerable<string> returnQ)
    {
        var intsToStrings = from i in ints
                            where i < 4
                            select i.ToString();

        returnQ = intsToStrings;
    }

    static void Main()
    {
        MQ app = new MQ();

        int[] nums = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };

        // QueryMethod1 returns a query as the value of the method.
        var myQuery1 = app.QueryMethod1(ref nums);

        // Query myQuery1 is executed in the following foreach loop.
        Console.WriteLine("Results of executing myQuery1:");
        // Rest the mouse pointer over myQuery1 to see its type.
        foreach (string s in myQuery1)
        {
            Console.WriteLine(s);
        }

        // You also can execute the query returned from QueryMethod1
        // myQuery1.Execute().ForEach(Console.WriteLine);
    }
}
```

```

// directly, without using myQuery1.
Console.WriteLine("\nResults of executing myQuery1 directly:");
// Rest the mouse pointer over the call to QueryMethod1 to see its
// return type.
foreach (string s in app.QueryMethod1(ref nums))
{
    Console.WriteLine(s);
}

IEnumerable<string> myQuery2;
// QueryMethod2 returns a query as the value of its out parameter.
app.QueryMethod2(ref nums, out myQuery2);

// Execute the returned query.
Console.WriteLine("\nResults of executing myQuery2:");
foreach (string s in myQuery2)
{
    Console.WriteLine(s);
}

// You can modify a query by using query composition. A saved query
// is nested inside a new query definition that revises the results
// of the first query.
myQuery1 = from item in myQuery1
           orderby item descending
           select item;

// Execute the modified query.
Console.WriteLine("\nResults of executing modified myQuery1:");
foreach (string s in myQuery1)
{
    Console.WriteLine(s);
}

// Keep console window open in debug mode.
Console.WriteLine("Press any key to exit.");
Console.ReadKey();
    }
}

```

See also

- [Language Integrated Query \(LINQ\)](#)

Store the results of a query in memory

12/28/2021 • 2 minutes to read • [Edit Online](#)

A query is basically a set of instructions for how to retrieve and organize data. Queries are executed lazily, as each subsequent item in the result is requested. When you use `foreach` to iterate the results, items are returned as accessed. To evaluate a query and store its results without executing a `foreach` loop, just call one of the following methods on the query variable:

- [ToList](#)
- [ToArray](#)
- [ToDictionary](#)
- [ToLookup](#)

We recommend that when you store the query results, you assign the returned collection object to a new variable as shown in the following example:

Example

```
class StoreQueryResults
{
    static List<int> numbers = new List<int>() { 1, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20 };
    static void Main()
    {
        IEnumerable<int> queryFactorsOfFour =
            from num in numbers
            where num % 4 == 0
            select num;

        // Store the results in a new variable
        // without executing a foreach loop.
        List<int> factorsofFourList = queryFactorsOfFour.ToList();

        // Iterate the list just to prove it holds data.
        Console.WriteLine(factorsofFourList[2]);
        factorsofFourList[2] = 0;
        Console.WriteLine(factorsofFourList[2]);

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key");
        Console.ReadKey();
    }
}
```

See also

- [Language Integrated Query \(LINQ\)](#)

Group query results

12/28/2021 • 8 minutes to read • [Edit Online](#)

Grouping is one of the most powerful capabilities of LINQ. The following examples show how to group data in various ways:

- By a single property.
- By the first letter of a string property.
- By a computed numeric range.
- By Boolean predicate or other expression.
- By a compound key.

In addition, the last two queries project their results into a new anonymous type that contains only the student's first and last name. For more information, see the [group clause](#).

Example helper class and data source

All the examples in this topic use the following helper classes and data sources.

```
public class StudentClass
{
    #region data
    protected enum GradeLevel { FirstYear = 1, SecondYear, ThirdYear, FourthYear };
    protected class Student
    {
        public string FirstName { get; set; }
        public string LastName { get; set; }
        public int ID { get; set; }
        public GradeLevel Year;
        public List<int> ExamScores;
    }

    protected static List<Student> students = new List<Student>
    {
        new Student {FirstName = "Terry", LastName = "Adams", ID = 120,
            Year = GradeLevel.SecondYear,
            ExamScores = new List<int>{ 99, 82, 81, 79}},
        new Student {FirstName = "Fadi", LastName = "Fakhouri", ID = 116,
            Year = GradeLevel.ThirdYear,
            ExamScores = new List<int>{ 99, 86, 90, 94}},
        new Student {FirstName = "Hanying", LastName = "Feng", ID = 117,
            Year = GradeLevel.FirstYear,
            ExamScores = new List<int>{ 93, 92, 80, 87}},
        new Student {FirstName = "Cesar", LastName = "Garcia", ID = 114,
            Year = GradeLevel.FourthYear,
            ExamScores = new List<int>{ 97, 89, 85, 82}},
        new Student {FirstName = "Debra", LastName = "Garcia", ID = 115,
            Year = GradeLevel.ThirdYear,
            ExamScores = new List<int>{ 35, 72, 91, 70}},
        new Student {FirstName = "Hugo", LastName = "Garcia", ID = 118,
            Year = GradeLevel.SecondYear,
            ExamScores = new List<int>{ 92, 90, 83, 78}},
        new Student {FirstName = "Sven", LastName = "Mortensen", ID = 113,
            Year = GradeLevel.FirstYear,
            ExamScores = new List<int>{ 88, 94, 65, 91}},
        new Student {FirstName = "Claire", LastName = "O'Donnell", ID = 112,
```

```

        Year = GradeLevel.FourthYear,
        ExamScores = new List<int>{ 75, 84, 91, 39}},
new Student {FirstName = "Svetlana", LastName = "Omelchenko", ID = 111,
    Year = GradeLevel.SecondYear,
    ExamScores = new List<int>{ 97, 92, 81, 60}},
new Student {FirstName = "Lance", LastName = "Tucker", ID = 119,
    Year = GradeLevel.ThirdYear,
    ExamScores = new List<int>{ 68, 79, 88, 92}},
new Student {FirstName = "Michael", LastName = "Tucker", ID = 122,
    Year = GradeLevel.FirstYear,
    ExamScores = new List<int>{ 94, 92, 91, 91}},
new Student {FirstName = "Eugene", LastName = "Zabokritski", ID = 121,
    Year = GradeLevel.FourthYear,
    ExamScores = new List<int>{ 96, 85, 91, 60}}
};
#endregion

//Helper method, used in GroupByRange.
protected static int GetPercentile(Student s)
{
    double avg = s.ExamScores.Average();
    return avg > 0 ? (int)avg / 10 : 0;
}

public void QueryHighScores(int exam, int score)
{
    var highScores = from student in students
        where student.ExamScores[exam] > score
        select new {Name = student.FirstName, Score = student.ExamScores[exam]};

    foreach (var item in highScores)
    {
        Console.WriteLine($"{item.Name,-15}{item.Score}");
    }
}

public class Program
{
    public static void Main()
    {
        StudentClass sc = new StudentClass();
        sc.QueryHighScores(1, 90);

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit");
        Console.ReadKey();
    }
}

```

Group by single property example

The following example shows how to group source elements by using a single property of the element as the group key. In this case the key is a `string`, the student's last name. It is also possible to use a substring for the key. The grouping operation uses the default equality comparer for the type.

Paste the following method into the `StudentClass` class. Change the calling statement in the `Main` method to `sc.GroupBySingleProperty()`.

```

public void GroupBySingleProperty()
{
    Console.WriteLine("Group by a single property in an object:");

    // Variable queryLastNames is an IEnumerable<IGrouping<string,
    // DataClass.Student>>.
    var queryLastNames =
        from student in students
        group student by student.LastName into newGroup
        orderby newGroup.Key
        select newGroup;

    foreach (var nameGroup in queryLastNames)
    {
        Console.WriteLine($"Key: {nameGroup.Key}");
        foreach (var student in nameGroup)
        {
            Console.WriteLine($"\\t{student.LastName}, {student.FirstName}");
        }
    }
}
/* Output:
Group by a single property in an object:
Key: Adams
    Adams, Terry
Key: Fakhouri
    Fakhouri, Fadi
Key: Feng
    Feng, Hanying
Key: Garcia
    Garcia, Cesar
    Garcia, Debra
    Garcia, Hugo
Key: Mortensen
    Mortensen, Sven
Key: O'Donnell
    O'Donnell, Claire
Key: Omelchenko
    Omelchenko, Svetlana
Key: Tucker
    Tucker, Lance
    Tucker, Michael
Key: Zabokritski
    Zabokritski, Eugene
*/

```

Group by value example

The following example shows how to group source elements by using something other than a property of the object for the group key. In this example, the key is the first letter of the student's last name.

Paste the following method into the `StudentClass` class. Change the calling statement in the `Main` method to `sc.GroupBySubstring()`.

```

public void GroupBySubstring()
{
    Console.WriteLine("\r\nGroup by something other than a property of the object:");

    var queryFirstLetters =
        from student in students
        group student by student.LastName[0];

    foreach (var studentGroup in queryFirstLetters)
    {
        Console.WriteLine($"Key: {studentGroup.Key}");
        // Nested foreach is required to access group items.
        foreach (var student in studentGroup)
        {
            Console.WriteLine($"  \t{student.LastName}, {student.FirstName}");
        }
    }
}
/* Output:
Group by something other than a property of the object:
Key: A
    Adams, Terry
Key: F
    Fakhouri, Fadi
    Feng, Hanying
Key: G
    Garcia, Cesar
    Garcia, Debra
    Garcia, Hugo
Key: M
    Mortensen, Sven
Key: O
    O'Donnell, Claire
    Omelchenko, Svetlana
Key: T
    Tucker, Lance
    Tucker, Michael
Key: Z
    Zabokritski, Eugene
*/

```

Group by a range example

The following example shows how to group source elements by using a numeric range as a group key. The query then projects the results into an anonymous type that contains only the first and last name and the percentile range to which the student belongs. An anonymous type is used because it is not necessary to use the complete `Student` object to display the results. `GetPercentile` is a helper function that calculates a percentile based on the student's average score. The method returns an integer between 0 and 10.

```

//Helper method, used in GroupByRange.
protected static int GetPercentile(Student s)
{
    double avg = s.ExamScores.Average();
    return avg > 0 ? (int)avg / 10 : 0;
}

```

Paste the following method into the `StudentClass` class. Change the calling statement in the `Main` method to `sc.GroupByRange()`.

```

public void GroupByRange()
{
    Console.WriteLine("\r\nGroup by numeric range and project into a new anonymous type:");

    var queryNumericRange =
        from student in students
        let percentile = GetPercentile(student)
        group new { student.FirstName, student.LastName } by percentile into percentGroup
        orderby percentGroup.Key
        select percentGroup;

    // Nested foreach required to iterate over groups and group items.
    foreach (var studentGroup in queryNumericRange)
    {
        Console.WriteLine($"Key: {studentGroup.Key * 10}");
        foreach (var item in studentGroup)
        {
            Console.WriteLine($"{item.LastName}, {item.FirstName}");
        }
    }
}
/* Output:
Group by numeric range and project into a new anonymous type:
Key: 60
    Garcia, Debra
Key: 70
    O'Donnell, Claire
Key: 80
    Adams, Terry
    Feng, Hanying
    Garcia, Cesar
    Garcia, Hugo
    Mortensen, Sven
    Omelchenko, Svetlana
    Tucker, Lance
    Zabokritski, Eugene
Key: 90
    Fakhouri, Fadi
    Tucker, Michael
*/

```

Group by comparison example

The following example shows how to group source elements by using a Boolean comparison expression. In this example, the Boolean expression tests whether a student's average exam score is greater than 75. As in previous examples, the results are projected into an anonymous type because the complete source element is not needed. Note that the properties in the anonymous type become properties on the `key` member and can be accessed by name when the query is executed.

Paste the following method into the `StudentClass` class. Change the calling statement in the `Main` method to `sc.GroupByBoolean()`.

```

public void GroupByBoolean()
{
    Console.WriteLine("\r\nGroup by a Boolean into two groups with string keys");
    Console.WriteLine("\True\" and \"False\" and project into a new anonymous type:");
    var queryGroupByAverages = from student in students
                                group new { student.FirstName, student.LastName }
                                    by student.ExamScores.Average() > 75 into studentGroup
                                select studentGroup;

    foreach (var studentGroup in queryGroupByAverages)
    {
        Console.WriteLine($"Key: {studentGroup.Key}");
        foreach (var student in studentGroup)
            Console.WriteLine($"{student.FirstName} {student.LastName}");
    }
}
/* Output:
Group by a Boolean into two groups with string keys
"True" and "False" and project into a new anonymous type:
Key: True
    Terry Adams
    Fadi Fakhouri
    Hanying Feng
    Cesar Garcia
    Hugo Garcia
    Sven Mortensen
    Svetlana Omelchenko
    Lance Tucker
    Michael Tucker
    Eugene Zabokritski
Key: False
    Debra Garcia
    Claire O'Donnell
*/

```

Group by anonymous type

The following example shows how to use an anonymous type to encapsulate a key that contains multiple values. In this example, the first key value is the first letter of the student's last name. The second key value is a Boolean that specifies whether the student scored over 85 on the first exam. You can order the groups by any property in the key.

Paste the following method into the `StudentClass` class. Change the calling statement in the `Main` method to `sc.GroupByCompositeKey()`.

```

public void GroupByCompositeKey()
{
    var queryHighScoreGroups =
        from student in students
        group student by new { FirstLetter = student.LastName[0],
                               Score = student.ExamScores[0] > 85 } into studentGroup
        orderby studentGroup.Key.FirstLetter
        select studentGroup;

    Console.WriteLine("\r\nGroup and order by a compound key:");
    foreach (var scoreGroup in queryHighScoreGroups)
    {
        string s = scoreGroup.Key.Score == true ? "more than" : "less than";
        Console.WriteLine($"Name starts with {scoreGroup.Key.FirstLetter} who scored {s} 85");
        foreach (var item in scoreGroup)
        {
            Console.WriteLine($"    {item.FirstName} {item.LastName}");
        }
    }
}

/* Output:
    Group and order by a compound key:
    Name starts with A who scored more than 85
        Terry Adams
    Name starts with F who scored more than 85
        Fadi Fakhouri
        Hanying Feng
    Name starts with G who scored more than 85
        Cesar Garcia
        Hugo Garcia
    Name starts with G who scored less than 85
        Debra Garcia
    Name starts with M who scored more than 85
        Sven Mortensen
    Name starts with O who scored less than 85
        Claire O'Donnell
    Name starts with O who scored more than 85
        Svetlana Omelchenko
    Name starts with T who scored less than 85
        Lance Tucker
    Name starts with T who scored more than 85
        Michael Tucker
    Name starts with Z who scored more than 85
        Eugene Zabokritski
*/

```

See also

- [GroupBy](#)
- [IGrouping<TKey,TElement>](#)
- [Language Integrated Query \(LINQ\)](#)
- [group clause](#)
- [Anonymous Types](#)
- [Perform a Subquery on a Grouping Operation](#)
- [Create a Nested Group](#)
- [Grouping Data](#)

Create a nested group

12/28/2021 • 2 minutes to read • [Edit Online](#)

The following example shows how to create nested groups in a LINQ query expression. Each group that is created according to student year or grade level is then further subdivided into groups based on the individuals' names.

Example

NOTE

This example contains references to objects that are defined in the sample code in [Query a collection of objects](#).


```

public void QueryNestedGroups()
{
    var queryNestedGroups =
        from student in students
        group student by student.Year into newGroup1
        from newGroup2 in
            (from student in newGroup1
             group student by student.LastName)
        group newGroup2 by newGroup1.Key;

    // Three nested foreach loops are required to iterate
    // over all elements of a grouped group. Hover the mouse
    // cursor over the iteration variables to see their actual type.
    foreach (var outerGroup in queryNestedGroups)
    {
        Console.WriteLine($"DataClass.Student Level = {outerGroup.Key}");
        foreach (var innerGroup in outerGroup)
        {
            Console.WriteLine($"\\tNames that begin with: {innerGroup.Key}");
            foreach (var innerGroupElement in innerGroup)
            {
                Console.WriteLine($"\\t\\t{innerGroupElement.LastName} {innerGroupElement.FirstName}");
            }
        }
    }
}
/*
Output:
DataClass.Student Level = SecondYear
    Names that begin with: Adams
        Adams Terry
    Names that begin with: Garcia
        Garcia Hugo
    Names that begin with: Omelchenko
        Omelchenko Svetlana
DataClass.Student Level = ThirdYear
    Names that begin with: Fakhouri
        Fakhouri Fadi
    Names that begin with: Garcia
        Garcia Debra
    Names that begin with: Tucker
        Tucker Lance
DataClass.Student Level = FirstYear
    Names that begin with: Feng
        Feng Hanying
    Names that begin with: Mortensen
        Mortensen Sven
    Names that begin with: Tucker
        Tucker Michael
DataClass.Student Level = FourthYear
    Names that begin with: Garcia
        Garcia Cesar
    Names that begin with: O'Donnell
        O'Donnell Claire
    Names that begin with: Zabokritski
        Zabokritski Eugene
*/

```

Note that three nested `foreach` loops are required to iterate over the inner elements of a nested group.

See also

- [Language Integrated Query \(LINQ\)](#)

Perform a subquery on a grouping operation

12/28/2021 • 2 minutes to read • [Edit Online](#)

This article shows two different ways to create a query that orders the source data into groups, and then performs a subquery over each group individually. The basic technique in each example is to group the source elements by using a *continuation* named `newGroup`, and then generating a new subquery against `newGroup`. This subquery is run against each new group that is created by the outer query. Note that in this particular example the final output is not a group, but a flat sequence of anonymous types.

For more information about how to group, see [group clause](#).

For more information about continuations, see [into](#). The following example uses an in-memory data structure as the data source, but the same principles apply for any kind of LINQ data source.

Example

NOTE

This example contains references to objects that are defined in the sample code in [Query a collection of objects](#).

```
public void QueryMax()
{
    var queryGroupMax =
        from student in students
        group student by student.Year into studentGroup
        select new
        {
            Level = studentGroup.Key,
            HighestScore =
                (from student2 in studentGroup
                 select student2.ExamScores.Average()).Max()
        };

    int count = queryGroupMax.Count();
    Console.WriteLine($"Number of groups = {count}");

    foreach (var item in queryGroupMax)
    {
        Console.WriteLine($" {item.Level} Highest Score={item.HighestScore}");
    }
}
```

The query in the snippet above can also be written using method syntax. The following code snippet has a semantically equivalent query written using method syntax.

```
public void QueryMaxUsingMethodSyntax()
{
    var queryGroupMax = students
        .GroupBy(student => student.Year)
        .Select(studentGroup => new
        {
            Level = studentGroup.Key,
            HighestScore = studentGroup.Select(student2 => student2.ExamScores.Average()).Max()
        });

    int count = queryGroupMax.Count();
    Console.WriteLine($"Number of groups = {count}");

    foreach (var item in queryGroupMax)
    {
        Console.WriteLine($" {item.Level} Highest Score={item.HighestScore}");
    }
}
```

See also

- [Language Integrated Query \(LINQ\)](#)

Group results by contiguous keys

12/28/2021 • 7 minutes to read • [Edit Online](#)

The following example shows how to group elements into chunks that represent subsequences of contiguous keys. For example, assume that you are given the following sequence of key-value pairs:

KEY	VALUE
A	We
A	think
A	that
B	Linq
C	is
A	really
B	cool
B	!

The following groups will be created in this order:

1. We, think, that
2. Linq
3. is
4. really
5. cool, !

The solution is implemented as an extension method that is thread-safe and that returns its results in a streaming manner. In other words, it produces its groups as it moves through the source sequence. Unlike the `group` or `orderby` operators, it can begin returning groups to the caller before all of the sequence has been read.

Thread-safety is accomplished by making a copy of each group or chunk as the source sequence is iterated, as explained in the source code comments. If the source sequence has a large sequence of contiguous items, the common language runtime may throw an [OutOfMemoryException](#).

Example

The following example shows both the extension method and the client code that uses it:

```
using System;
using System.Collections.Generic;
using System.Linq;
```

```

namespace ChunkIt
{
    // Static class to contain the extension methods.
    public static class MyExtensions
    {
        public static IEnumerable<IGrouping<TKey, TSource>> ChunkBy<TSource, TKey>(this IEnumerable<TSource>
source, Func<TSource, TKey> keySelector)
        {
            return source.ChunkBy(keySelector, EqualityComparer<TKey>.Default);
        }

        public static IEnumerable<IGrouping<TKey, TSource>> ChunkBy<TSource, TKey>(this IEnumerable<TSource>
source, Func<TSource, TKey> keySelector, IEqualityComparer<TKey> comparer)
        {
            // Flag to signal end of source sequence.
            const bool noMoreSourceElements = true;

            // Auto-generated iterator for the source array.
            var enumerator = source.GetEnumerator();

            // Move to the first element in the source sequence.
            if (!enumerator.MoveNext()) yield break;

            // Iterate through source sequence and create a copy of each Chunk.
            // On each pass, the iterator advances to the first element of the next "Chunk"
            // in the source sequence. This loop corresponds to the outer foreach loop that
            // executes the query.
            Chunk<TKey, TSource> current = null;
            while (true)
            {
                // Get the key for the current Chunk. The source iterator will churn through
                // the source sequence until it finds an element with a key that doesn't match.
                var key = keySelector(enumerator.Current);

                // Make a new Chunk (group) object that initially has one GroupItem, which is a copy of the
                current source element.
                current = new Chunk<TKey, TSource>(key, enumerator, value => comparer.Equals(key,
keySelector(value)));

                // Return the Chunk. A Chunk is an IGrouping<TKey,TSource>, which is the return value of the
                ChunkBy method.
                // At this point the Chunk only has the first element in its source sequence. The remaining
                elements will be
                // returned only when the client code foreach's over this chunk. See Chunk.GetEnumerator for
                more info.
                yield return current;

                // Check to see whether (a) the chunk has made a copy of all its source elements or
                // (b) the iterator has reached the end of the source sequence. If the caller uses an inner
                // foreach loop to iterate the chunk items, and that loop ran to completion,
                // then the Chunk.GetEnumerator method will already have made
                // copies of all chunk items before we get here. If the Chunk.GetEnumerator loop did not
                // enumerate all elements in the chunk, we need to do it here to avoid corrupting the
                iterator
                // for clients that may be calling us on a separate thread.
                if (current.CopyAllChunkElements() == noMoreSourceElements)
                {
                    yield break;
                }
            }
        }

        // A Chunk is a contiguous group of one or more source elements that have the same key. A Chunk
        // has a key and a list of ChunkItem objects, which are copies of the elements in the source
        sequence.
        class Chunk<TKey, TSource> : IGrouping<TKey, TSource>
        {
            // INVARIANT: DoneCopyingChunk == true ||
            // (predicate != null && predicate(enumerator.Current) && current.Value == enumerator.Current)

```

```

        // A Chunk has a linked list of ChunkItems, which represent the elements in the current chunk.
Each ChunkItem
    // has a reference to the next ChunkItem in the list.
    class ChunkItem
    {
        public ChunkItem(TSource value)
        {
            Value = value;
        }
        public readonly TSource Value;
        public ChunkItem Next = null;
    }

    // The value that is used to determine matching elements
    private readonly TKey key;

    // Stores a reference to the enumerator for the source sequence
    private IEnumerator<TSource> enumerator;

    // A reference to the predicate that is used to compare keys.
    private Func<TSource, bool> predicate;

    // Stores the contents of the first source element that
    // belongs with this chunk.
    private readonly ChunkItem head;

    // End of the list. It is repositioned each time a new
    // ChunkItem is added.
    private ChunkItem tail;

    // Flag to indicate the source iterator has reached the end of the source sequence.
    internal bool isLastSourceElement = false;

    // Private object for thread synchronization
    private object m_Lock;

    // REQUIRES: enumerator != null && predicate != null
    public Chunk(TKey key, IEnumerator<TSource> enumerator, Func<TSource, bool> predicate)
    {
        this.key = key;
        this.enumerator = enumerator;
        this.predicate = predicate;

        // A Chunk always contains at least one element.
        head = new ChunkItem(enumerator.Current);

        // The end and beginning are the same until the list contains > 1 elements.
        tail = head;

        m_Lock = new object();
    }

    // Indicates that all chunk elements have been copied to the list of ChunkItems,
    // and the source enumerator is either at the end, or else on an element with a new key.
    // the tail of the linked list is set to null in the CopyNextChunkElement method if the
    // key of the next element does not match the current chunk's key, or there are no more elements
in the source.
    private bool DoneCopyingChunk => tail == null;

    // Adds one ChunkItem to the current group
    // REQUIRES: !DoneCopyingChunk && lock(this)
    private void CopyNextChunkElement()
    {
        // Try to advance the iterator on the source sequence.
        // If MoveNext returns false we are at the end, and isLastSourceElement is set to true
        isLastSourceElement = !enumerator.MoveNext();

        // If we are (a) at the end of the source, or (b) at the end of the current chunk

```

```

        // then null out the enumerator and predicate for reuse with the next chunk.
        if (isLastSourceElement || !predicate(enumerator.Current))
        {
            enumerator = null;
            predicate = null;
        }
        else
        {
            tail.Next = new ChunkItem(enumerator.Current);
        }

        // tail will be null if we are at the end of the chunk elements
        // This check is made in DoneCopyingChunk.
        tail = tail.Next;
    }

    // Called after the end of the last chunk was reached. It first checks whether
    // there are more elements in the source sequence. If there are, it
    // Returns true if enumerator for this chunk was exhausted.
    internal bool CopyAllChunkElements()
    {
        while (true)
        {
            lock (m_Lock)
            {
                if (DoneCopyingChunk)
                {
                    // If isLastSourceElement is false,
                    // it signals to the outer iterator
                    // to continue iterating.
                    return isLastSourceElement;
                }
                else
                {
                    CopyNextChunkElement();
                }
            }
        }
    }
}

public TKey Key => key;

// Invoked by the inner foreach loop. This method stays just one step ahead
// of the client requests. It adds the next element of the chunk only after
// the clients requests the last element in the list so far.
public IEnumerator<TSource> GetEnumerator()
{
    //Specify the initial element to enumerate.
    ChunkItem current = head;

    // There should always be at least one ChunkItem in a Chunk.
    while (current != null)
    {
        // Yield the current item in the list.
        yield return current.Value;

        // Copy the next item from the source sequence,
        // if we are at the end of our local list.
        lock (m_Lock)
        {
            if (current == tail)
            {
                CopyNextChunkElement();
            }
        }

        // Move to the next ChunkItem in the list.
        current = current.Next;
    }
}

```

```

    }

    System.Collections.IEnumerator System.Collections.IEnumerable.GetEnumerator() =>
GetEnumerator();
    }
}

// A simple named type is used for easier viewing in the debugger. Anonymous types
// work just as well with the ChunkBy operator.
public class KeyValPair
{
    public string Key { get; set; }
    public string Value { get; set; }
}

class Program
{
    // The source sequence.
    public static IEnumerable<KeyValPair> list;

    // Query variable declared as class member to be available
    // on different threads.
    static IEnumerable<IGrouping<string, KeyValPair>> query;

    static void Main(string[] args)
    {
        // Initialize the source sequence with an array initializer.
        list = new[]
        {
            new KeyValPair{ Key = "A", Value = "We" },
            new KeyValPair{ Key = "A", Value = "think" },
            new KeyValPair{ Key = "A", Value = "that" },
            new KeyValPair{ Key = "B", Value = "Linq" },
            new KeyValPair{ Key = "C", Value = "is" },
            new KeyValPair{ Key = "A", Value = "really" },
            new KeyValPair{ Key = "B", Value = "cool" },
            new KeyValPair{ Key = "B", Value = "!" }
        };

        // Create the query by using our user-defined query operator.
        query = list.ChunkBy(p => p.Key);

        // ChunkBy returns IGrouping objects, therefore a nested
        // foreach loop is required to access the elements in each "chunk".
        foreach (var item in query)
        {
            Console.WriteLine($"Group key = {item.Key}");
            foreach (var inner in item)
            {
                Console.WriteLine($"{inner.Value}");
            }
        }

        Console.WriteLine("Press any key to exit");
        Console.ReadKey();
    }
}

```

To use the extension method in your project, copy the `MyExtensions` static class to a new or existing source code file and if it is required, add a `using` directive for the namespace where it is located.

See also

- [Language Integrated Query \(LINQ\)](#)

Dynamically specify predicate filters at run time

12/28/2021 • 2 minutes to read • [Edit Online](#)

In some cases, you don't know until run time how many predicates you have to apply to source elements in the `where` clause. One way to dynamically specify multiple predicate filters is to use the [Contains](#) method, as shown in the following example. The example is constructed in two ways. First, the project is run by filtering on values that are provided in the program. Then the project is run again by using input provided at run time.

To filter by using the Contains method

1. Open a new console application and name it `PredicateFilters`.
2. Copy the `StudentClass` class from [Query a collection of objects](#) and paste it into namespace `PredicateFilters` underneath class `Program`. `StudentClass` provides a list of `Student` objects.
3. Comment out the `Main` method in `StudentClass`.
4. Replace class `Program` with the following code:

```
class DynamicPredicates : StudentClass
{
    static void Main(string[] args)
    {
        string[] ids = { "111", "114", "112" };

        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }

    static void QueryByID(string[] ids)
    {
        var queryNames =
            from student in students
            let i = student.ID.ToString()
            where ids.Contains(i)
            select new { student.LastName, student.ID };

        foreach (var name in queryNames)
        {
            Console.WriteLine($"{name.LastName}: {name.ID}");
        }
    }
}
```

5. Add the following line to the `Main` method in class `DynamicPredicates`, under the declaration of `ids`.

```
QueryById(ids);
```

6. Run the project.
7. The following output is displayed in a console window:

Garcia: 114

O'Donnell: 112

Omelchenko: 111

8. The next step is to run the project again, this time by using input entered at run time instead of array `ids`. Change `QueryByID(ids)` to `QueryByID(args)` in the `Main` method.
9. Run the project with the command line arguments `122 117 120 115`. When the project is run, those values become elements of `args`, the parameter of the `Main` method.
10. The following output is displayed in a console window:

Adams: 120

Feng: 117

Garcia: 115

Tucker: 122

To filter by using a switch statement

1. You can use a `switch` statement to select among predetermined alternative queries. In the following example, `studentQuery` uses a different `where` clause depending on which grade level, or year, is specified at run time.
2. Copy the following method and paste it into class `DynamicPredicates`.

```

// To run this sample, first specify an integer value of 1 to 4 for the command
// line. This number will be converted to a GradeLevel value that specifies which
// set of students to query.
// Call the method: QueryByYear(args[0]);

static void QueryByYear(string level)
{
    GradeLevel year = (GradeLevel)Convert.ToInt32(level);
    IEnumerable<Student> studentQuery = null;
    switch (year)
    {
        case GradeLevel.FirstYear:
            studentQuery = from student in students
                           where student.Year == GradeLevel.FirstYear
                           select student;

            break;
        case GradeLevel.SecondYear:
            studentQuery = from student in students
                           where student.Year == GradeLevel.SecondYear
                           select student;

            break;
        case GradeLevel.ThirdYear:
            studentQuery = from student in students
                           where student.Year == GradeLevel.ThirdYear
                           select student;

            break;
        case GradeLevel.FourthYear:
            studentQuery = from student in students
                           where student.Year == GradeLevel.FourthYear
                           select student;

            break;

        default:
            break;
    }
    Console.WriteLine($"The following students are at level {year}");
    foreach (Student name in studentQuery)
    {
        Console.WriteLine($"{name.LastName}: {name.ID}");
    }
}

```

3. In the `Main` method, replace the call to `QueryByID` with the following call, which sends the first element from the `args` array as its argument: `QueryByYear(args[0])`.
4. Run the project with a command line argument of an integer value between 1 and 4.

See also

- [Language Integrated Query \(LINQ\)](#)
- [where clause](#)

Perform inner joins

12/28/2021 • 10 minutes to read • [Edit Online](#)

In relational database terms, an *inner join* produces a result set in which each element of the first collection appears one time for every matching element in the second collection. If an element in the first collection has no matching elements, it does not appear in the result set. The `Join` method, which is called by the `join` clause in C#, implements an inner join.

This article shows you how to perform four variations of an inner join:

- A simple inner join that correlates elements from two data sources based on a simple key.
- An inner join that correlates elements from two data sources based on a *composite* key. A composite key, which is a key that consists of more than one value, enables you to correlate elements based on more than one property.
- A *multiple join* in which successive join operations are appended to each other.
- An inner join that is implemented by using a group join.

Example - Simple key join

The following example creates two collections that contain objects of two user-defined types, `Person` and `Pet`. The query uses the `join` clause in C# to match `Person` objects with `Pet` objects whose `Owner` is that `Person`. The `select` clause in C# defines how the resulting objects will look. In this example the resulting objects are anonymous types that consist of the owner's first name and the pet's name.

```

class Person
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
}

class Pet
{
    public string Name { get; set; }
    public Person Owner { get; set; }
}

/// <summary>
/// Simple inner join.
/// </summary>
public static void InnerJoinExample()
{
    Person magnus = new Person { FirstName = "Magnus", LastName = "Hedlund" };
    Person terry = new Person { FirstName = "Terry", LastName = "Adams" };
    Person charlotte = new Person { FirstName = "Charlotte", LastName = "Weiss" };
    Person arlene = new Person { FirstName = "Arlene", LastName = "Huff" };
    Person rui = new Person { FirstName = "Rui", LastName = "Raposo" };

    Pet barley = new Pet { Name = "Barley", Owner = terry };
    Pet boots = new Pet { Name = "Boots", Owner = terry };
    Pet whiskers = new Pet { Name = "Whiskers", Owner = charlotte };
    Pet bluemoon = new Pet { Name = "Blue Moon", Owner = rui };
    Pet daisy = new Pet { Name = "Daisy", Owner = magnus };

    // Create two lists.
    List<Person> people = new List<Person> { magnus, terry, charlotte, arlene, rui };
    List<Pet> pets = new List<Pet> { barley, boots, whiskers, bluemoon, daisy };

    // Create a collection of person-pet pairs. Each element in the collection
    // is an anonymous type containing both the person's name and their pet's name.
    var query = from person in people
                join pet in pets on person equals pet.Owner
                select new { OwnerName = person.FirstName, PetName = pet.Name };

    foreach (var ownerAndPet in query)
    {
        Console.WriteLine($"{ownerAndPet.PetName}\" is owned by {ownerAndPet.OwnerName}");
    }
}

// This code produces the following output:
//
// "Daisy" is owned by Magnus
// "Barley" is owned by Terry
// "Boots" is owned by Terry
// "Whiskers" is owned by Charlotte
// "Blue Moon" is owned by Rui

```

Note that the `Person` object whose `LastName` is "Huff" does not appear in the result set because there is no `Pet` object that has `Pet.Owner` equal to that `Person`.

Example - Composite key join

Instead of correlating elements based on just one property, you can use a composite key to compare elements based on multiple properties. To do this, specify the key selector function for each collection to return an anonymous type that consists of the properties you want to compare. If you label the properties, they must have the same label in each key's anonymous type. The properties must also appear in the same order.

The following example uses a list of `Employee` objects and a list of `Student` objects to determine which

employees are also students. Both of these types have a `FirstName` and a `LastName` property of type `String`. The functions that create the join keys from each list's elements return an anonymous type that consists of the `FirstName` and `LastName` properties of each element. The join operation compares these composite keys for equality and returns pairs of objects from each list where both the first name and the last name match.

```
class Employee
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public int EmployeeID { get; set; }
}

class Student
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public int StudentID { get; set; }
}

/// <summary>
/// Performs a join operation using a composite key.
/// </summary>
public static void CompositeKeyJoinExample()
{
    // Create a list of employees.
    List<Employee> employees = new List<Employee> {
        new Employee { FirstName = "Terry", LastName = "Adams", EmployeeID = 522459 },
        new Employee { FirstName = "Charlotte", LastName = "Weiss", EmployeeID = 204467 },
        new Employee { FirstName = "Magnus", LastName = "Hedland", EmployeeID = 866200 },
        new Employee { FirstName = "Vernette", LastName = "Price", EmployeeID = 437139 } };

    // Create a list of students.
    List<Student> students = new List<Student> {
        new Student { FirstName = "Vernette", LastName = "Price", StudentID = 9562 },
        new Student { FirstName = "Terry", LastName = "Earls", StudentID = 9870 },
        new Student { FirstName = "Terry", LastName = "Adams", StudentID = 9913 } };

    // Join the two data sources based on a composite key consisting of first and last name,
    // to determine which employees are also students.
    IEnumerable<string> query = from employee in employees
                                join student in students
                                on new { employee.FirstName, employee.LastName }
                                equals new { student.FirstName, student.LastName }
                                select employee.FirstName + " " + employee.LastName;

    Console.WriteLine("The following people are both employees and students:");
    foreach (string name in query)
        Console.WriteLine(name);
}

// This code produces the following output:
//
// The following people are both employees and students:
// Terry Adams
// Vernette Price
```

Example - Multiple join

Any number of join operations can be appended to each other to perform a multiple join. Each `join` clause in C# correlates a specified data source with the results of the previous join.

The following example creates three collections: a list of `Person` objects, a list of `Cat` objects, and a list of `Dog` objects.

The first `join` clause in C# matches people and cats based on a `Person` object matching `Cat.Owner`. It returns a sequence of anonymous types that contain the `Person` object and `Cat.Name`.

The second `join` clause in C# correlates the anonymous types returned by the first join with `Dog` objects in the supplied list of dogs, based on a composite key that consists of the `Owner` property of type `Person`, and the first letter of the animal's name. It returns a sequence of anonymous types that contain the `Cat.Name` and `Dog.Name` properties from each matching pair. Because this is an inner join, only those objects from the first data source that have a match in the second data source are returned.

```
class Person
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
}

class Pet
{
    public string Name { get; set; }
    public Person Owner { get; set; }
}

class Cat : Pet
{ }

class Dog : Pet
{ }

public static void MultipleJoinExample()
{
    Person magnus = new Person { FirstName = "Magnus", LastName = "Hedlund" };
    Person terry = new Person { FirstName = "Terry", LastName = "Adams" };
    Person charlotte = new Person { FirstName = "Charlotte", LastName = "Weiss" };
    Person arlene = new Person { FirstName = "Arlene", LastName = "Huff" };
    Person rui = new Person { FirstName = "Rui", LastName = "Raposo" };
    Person phyllis = new Person { FirstName = "Phyllis", LastName = "Harris" };

    Cat barley = new Cat { Name = "Barley", Owner = terry };
    Cat boots = new Cat { Name = "Boots", Owner = terry };
    Cat whiskers = new Cat { Name = "Whiskers", Owner = charlotte };
    Cat bluemoon = new Cat { Name = "Blue Moon", Owner = rui };
    Cat daisy = new Cat { Name = "Daisy", Owner = magnus };

    Dog fourwheeldrive = new Dog { Name = "Four Wheel Drive", Owner = phyllis };
    Dog duke = new Dog { Name = "Duke", Owner = magnus };
    Dog denim = new Dog { Name = "Denim", Owner = terry };
    Dog wiley = new Dog { Name = "Wiley", Owner = charlotte };
    Dog snoopy = new Dog { Name = "Snoopy", Owner = rui };
    Dog snickers = new Dog { Name = "Snickers", Owner = arlene };

    // Create three lists.
    List<Person> people =
        new List<Person> { magnus, terry, charlotte, arlene, rui, phyllis };
    List<Cat> cats =
        new List<Cat> { barley, boots, whiskers, bluemoon, daisy };
    List<Dog> dogs =
        new List<Dog> { fourwheeldrive, duke, denim, wiley, snoopy, snickers };

    // The first join matches Person and Cat.Owner from the list of people and
    // cats, based on a common Person. The second join matches dogs whose names start
    // with the same letter as the cats that have the same owner.
    var query = from person in people
                join cat in cats on person equals cat.Owner
                join dog in dogs on
                    new { Owner = person, Letter = cat.Name.Substring(0, 1) }
                    equals new { dog.Owner, Letter = dog.Name.Substring(0, 1) }
                select new { CatName = cat.Name, DogName = dog.Name };
```

```

    foreach (var obj in query)
    {
        Console.WriteLine(
            $"The cat \"{obj.CatName}\" shares a house, and the first letter of their name, with \"
{obj.DogName}\".");
    }
}

// This code produces the following output:
//
// The cat "Daisy" shares a house, and the first letter of their name, with "Duke".
// The cat "Whiskers" shares a house, and the first letter of their name, with "Wiley".

```

Example - Inner join by using grouped join

The following example shows you how to implement an inner join by using a group join.

In `query1`, the list of `Person` objects is group-joined to the list of `Pet` objects based on the `Person` matching the `Pet.Owner` property. The group join creates a collection of intermediate groups, where each group consists of a `Person` object and a sequence of matching `Pet` objects.

By adding a second `from` clause to the query, this sequence of sequences is combined (or flattened) into one longer sequence. The type of the elements of the final sequence is specified by the `select` clause. In this example, that type is an anonymous type that consists of the `Person.FirstName` and `Pet.Name` properties for each matching pair.

The result of `query1` is equivalent to the result set that would have been obtained by using the `join` clause without the `into` clause to perform an inner join. The `query2` variable demonstrates this equivalent query.

```

class Person
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
}

class Pet
{
    public string Name { get; set; }
    public Person Owner { get; set; }
}

/// <summary>
/// Performs an inner join by using GroupJoin().
/// </summary>
public static void InnerGroupJoinExample()
{
    Person magnus = new Person { FirstName = "Magnus", LastName = "Hedlund" };
    Person terry = new Person { FirstName = "Terry", LastName = "Adams" };
    Person charlotte = new Person { FirstName = "Charlotte", LastName = "Weiss" };
    Person arlene = new Person { FirstName = "Arlene", LastName = "Huff" };

    Pet barley = new Pet { Name = "Barley", Owner = terry };
    Pet boots = new Pet { Name = "Boots", Owner = terry };
    Pet whiskers = new Pet { Name = "Whiskers", Owner = charlotte };
    Pet bluemoon = new Pet { Name = "Blue Moon", Owner = terry };
    Pet daisy = new Pet { Name = "Daisy", Owner = magnus };

    // Create two lists.
    List<Person> people = new List<Person> { magnus, terry, charlotte, arlene };
    List<Pet> pets = new List<Pet> { barley, boots, whiskers, bluemoon, daisy };

    var query1 = from person in people

```



```

        join pet in pets on person equals pet.Owner into gj
        from subpet in gj
        select new { OwnerName = person.FirstName, PetName = subpet.Name };

Console.WriteLine("Inner join using GroupJoin()");
foreach (var v in query1)
{
    Console.WriteLine($"{v.OwnerName} - {v.PetName}");
}

var query2 = from person in people
              join pet in pets on person equals pet.Owner
              select new { OwnerName = person.FirstName, PetName = pet.Name };

Console.WriteLine("\nThe equivalent operation using Join()");
foreach (var v in query2)
    Console.WriteLine($"{v.OwnerName} - {v.PetName}");
}

// This code produces the following output:
//
// Inner join using GroupJoin():
// Magnus - Daisy
// Terry - Barley
// Terry - Boots
// Terry - Blue Moon
// Charlotte - Whiskers
//
// The equivalent operation using Join():
// Magnus - Daisy
// Terry - Barley
// Terry - Boots
// Terry - Blue Moon
// Charlotte - Whiskers

```

See also

- [Join](#)
- [GroupJoin](#)
- [Perform grouped joins](#)
- [Perform left outer joins](#)
- [Anonymous types](#)

Perform grouped joins

12/28/2021 • 5 minutes to read • [Edit Online](#)

The group join is useful for producing hierarchical data structures. It pairs each element from the first collection with a set of correlated elements from the second collection.

For example, a class or a relational database table named `Student` might contain two fields: `Id` and `Name`. A second class or relational database table named `Course` might contain two fields: `StudentId` and `CourseTitle`. A group join of these two data sources, based on matching `Student.Id` and `Course.StudentId`, would group each `Student` with a collection of `Course` objects (which might be empty).

NOTE

Each element of the first collection appears in the result set of a group join regardless of whether correlated elements are found in the second collection. In the case where no correlated elements are found, the sequence of correlated elements for that element is empty. The result selector therefore has access to every element of the first collection. This differs from the result selector in a non-group join, which cannot access elements from the first collection that have no match in the second collection.

WARNING

`Enumerable.GroupJoin` has no direct equivalent in traditional relational database terms. However, this method does implement a superset of inner joins and left outer joins. Both of these operations can be written in terms of a grouped join. For more information, see [Join Operations](#) and [Entity Framework Core, GroupJoin](#).

The first example in this article shows you how to perform a group join. The second example shows you how to use a group join to create XML elements.

Example - Group join

The following example performs a group join of objects of type `Person` and `Pet` based on the `Person` matching the `Pet.Owner` property. Unlike a non-group join, which would produce a pair of elements for each match, the group join produces only one resulting object for each element of the first collection, which in this example is a `Person` object. The corresponding elements from the second collection, which in this example are `Pet` objects, are grouped into a collection. Finally, the result selector function creates an anonymous type for each match that consists of `Person.FirstName` and a collection of `Pet` objects.

```

class Person
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
}

class Pet
{
    public string Name { get; set; }
    public Person Owner { get; set; }
}

/// <summary>
/// This example performs a grouped join.
/// </summary>
public static void GroupJoinExample()
{
    Person magnus = new Person { FirstName = "Magnus", LastName = "Hedlund" };
    Person terry = new Person { FirstName = "Terry", LastName = "Adams" };
    Person charlotte = new Person { FirstName = "Charlotte", LastName = "Weiss" };
    Person arlene = new Person { FirstName = "Arlene", LastName = "Huff" };

    Pet barley = new Pet { Name = "Barley", Owner = terry };
    Pet boots = new Pet { Name = "Boots", Owner = terry };
    Pet whiskers = new Pet { Name = "Whiskers", Owner = charlotte };
    Pet bluemoon = new Pet { Name = "Blue Moon", Owner = terry };
    Pet daisy = new Pet { Name = "Daisy", Owner = magnus };

    // Create two lists.
    List<Person> people = new List<Person> { magnus, terry, charlotte, arlene };
    List<Pet> pets = new List<Pet> { barley, boots, whiskers, bluemoon, daisy };

    // Create a list where each element is an anonymous type
    // that contains the person's first name and a collection of
    // pets that are owned by them.
    var query = from person in people
                join pet in pets on person equals pet.Owner into gj
                select new { OwnerName = person.FirstName, Pets = gj };

    foreach (var v in query)
    {
        // Output the owner's name.
        Console.WriteLine($"{v.OwnerName}:");
        // Output each of the owner's pet's names.
        foreach (Pet pet in v.Pets)
            Console.WriteLine($"  {pet.Name}");
    }
}

// This code produces the following output:
//
// Magnus:
//   Daisy
// Terry:
//   Barley
//   Boots
//   Blue Moon
// Charlotte:
//   Whiskers
// Arlene:

```

Example - Group join to create XML

Group joins are ideal for creating XML by using LINQ to XML. The following example is similar to the previous example except that instead of creating anonymous types, the result selector function creates XML elements that

represent the joined objects.

```
class Person
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
}

class Pet
{
    public string Name { get; set; }
    public Person Owner { get; set; }
}

/// <summary>
/// This example creates XML output from a grouped join.
/// </summary>
public static void GroupJoinXMLExample()
{
    Person magnus = new Person { FirstName = "Magnus", LastName = "Hedlund" };
    Person terry = new Person { FirstName = "Terry", LastName = "Adams" };
    Person charlotte = new Person { FirstName = "Charlotte", LastName = "Weiss" };
    Person arlene = new Person { FirstName = "Arlene", LastName = "Huff" };

    Pet barley = new Pet { Name = "Barley", Owner = terry };
    Pet boots = new Pet { Name = "Boots", Owner = terry };
    Pet whiskers = new Pet { Name = "Whiskers", Owner = charlotte };
    Pet bluemoon = new Pet { Name = "Blue Moon", Owner = terry };
    Pet daisy = new Pet { Name = "Daisy", Owner = magnus };

    // Create two lists.
    List<Person> people = new List<Person> { magnus, terry, charlotte, arlene };
    List<Pet> pets = new List<Pet> { barley, boots, whiskers, bluemoon, daisy };

    // Create XML to display the hierarchical organization of people and their pets.
    XElement ownersAndPets = new XElement("PetOwners",
        from person in people
        join pet in pets on person equals pet.Owner into gj
        select new XElement("Person",
            new XAttribute("FirstName", person.FirstName),
            new XAttribute("LastName", person.LastName),
            from subpet in gj
            select new XElement("Pet", subpet.Name)));

    Console.WriteLine(ownersAndPets);
}

// This code produces the following output:
//
// <PetOwners>
//   <Person FirstName="Magnus" LastName="Hedlund">
//     <Pet>Daisy</Pet>
//   </Person>
//   <Person FirstName="Terry" LastName="Adams">
//     <Pet>Barley</Pet>
//     <Pet>Boots</Pet>
//     <Pet>Blue Moon</Pet>
//   </Person>
//   <Person FirstName="Charlotte" LastName="Weiss">
//     <Pet>Whiskers</Pet>
//   </Person>
//   <Person FirstName="Arlene" LastName="Huff" />
// </PetOwners>
```

See also

- [Join](#)
- [GroupJoin](#)
- [Perform inner joins](#)
- [Perform left outer joins](#)
- [Anonymous types](#)

Perform left outer joins

12/28/2021 • 2 minutes to read • [Edit Online](#)

A left outer join is a join in which each element of the first collection is returned, regardless of whether it has any correlated elements in the second collection. You can use LINQ to perform a left outer join by calling the [DefaultIfEmpty](#) method on the results of a group join.

Example

The following example demonstrates how to use the [DefaultIfEmpty](#) method on the results of a group join to perform a left outer join.

The first step in producing a left outer join of two collections is to perform an inner join by using a group join. (See [Perform inner joins](#) for an explanation of this process.) In this example, the list of `Person` objects is inner-joined to the list of `Pet` objects based on a `Person` object that matches `Pet.Owner`.

The second step is to include each element of the first (left) collection in the result set even if that element has no matches in the right collection. This is accomplished by calling [DefaultIfEmpty](#) on each sequence of matching elements from the group join. In this example, [DefaultIfEmpty](#) is called on each sequence of matching `Pet` objects. The method returns a collection that contains a single, default value if the sequence of matching `Pet` objects is empty for any `Person` object, thereby ensuring that each `Person` object is represented in the result collection.

NOTE

The default value for a reference type is `null`; therefore, the example checks for a null reference before accessing each element of each `Pet` collection.

```

class Person
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
}

class Pet
{
    public string Name { get; set; }
    public Person Owner { get; set; }
}

public static void LeftOuterJoinExample()
{
    Person magnus = new Person { FirstName = "Magnus", LastName = "Hedlund" };
    Person terry = new Person { FirstName = "Terry", LastName = "Adams" };
    Person charlotte = new Person { FirstName = "Charlotte", LastName = "Weiss" };
    Person arlene = new Person { FirstName = "Arlene", LastName = "Huff" };

    Pet barley = new Pet { Name = "Barley", Owner = terry };
    Pet boots = new Pet { Name = "Boots", Owner = terry };
    Pet whiskers = new Pet { Name = "Whiskers", Owner = charlotte };
    Pet bluemoon = new Pet { Name = "Blue Moon", Owner = terry };
    Pet daisy = new Pet { Name = "Daisy", Owner = magnus };

    // Create two lists.
    List<Person> people = new List<Person> { magnus, terry, charlotte, arlene };
    List<Pet> pets = new List<Pet> { barley, boots, whiskers, bluemoon, daisy };

    var query = from person in people
                join pet in pets on person equals pet.Owner into gj
                from subpet in gj.DefaultIfEmpty()
                select new { person.FirstName, PetName = subpet?.Name ?? String.Empty };

    foreach (var v in query)
    {
        Console.WriteLine($"{v.FirstName+":",-15}{v.PetName}");
    }
}

// This code produces the following output:
//
// Magnus:      Daisy
// Terry:       Barley
// Terry:       Boots
// Terry:       Blue Moon
// Charlotte:   Whiskers
// Arlene:

```

See also

- [Join](#)
- [GroupJoin](#)
- [Perform inner joins](#)
- [Perform grouped joins](#)
- [Anonymous types](#)

Order the results of a join clause

12/28/2021 • 2 minutes to read • [Edit Online](#)

This example shows how to order the results of a join operation. Note that the ordering is performed after the join. Although you can use an `orderby` clause with one or more of the source sequences before the join, generally we do not recommend it. Some LINQ providers might not preserve that ordering after the join.

Example

This query creates a group join, and then sorts the groups based on the category element, which is still in scope. Inside the anonymous type initializer, a sub-query orders all the matching elements from the products sequence.

```
class HowToOrderJoins
{
    #region Data
    class Product
    {
        public string Name { get; set; }
        public int CategoryID { get; set; }
    }

    class Category
    {
        public string Name { get; set; }
        public int ID { get; set; }
    }

    // Specify the first data source.
    List<Category> categories = new List<Category>()
{
    new Category(){Name="Beverages", ID=001},
    new Category(){ Name="Condiments", ID=002},
    new Category(){ Name="Vegetables", ID=003},
    new Category() { Name="Grains", ID=004},
    new Category() { Name="Fruit", ID=005}
};

    // Specify the second data source.
    List<Product> products = new List<Product>()
{
    new Product{Name="Cola", CategoryID=001},
    new Product{Name="Tea", CategoryID=001},
    new Product{Name="Mustard", CategoryID=002},
    new Product{Name="Pickles", CategoryID=002},
    new Product{Name="Carrots", CategoryID=003},
    new Product{Name="Bok Choy", CategoryID=003},
    new Product{Name="Peaches", CategoryID=005},
    new Product{Name="Melons", CategoryID=005},
};

    #endregion
    static void Main()
    {
        HowToOrderJoins app = new HowToOrderJoins();
        app.OrderJoin1();

        // Keep console window open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
```



```

void OrderJoin1()
{
    var groupJoinQuery2 =
        from category in categories
        join prod in products on category.ID equals prod.CategoryID into prodGroup
        orderby category.Name
        select new
        {
            Category = category.Name,
            Products = from prod2 in prodGroup
                       orderby prod2.Name
                       select prod2
        };

    foreach (var productGroup in groupJoinQuery2)
    {
        Console.WriteLine(productGroup.Category);
        foreach (var prodItem in productGroup.Products)
        {
            Console.WriteLine($" {prodItem.Name,-10} {prodItem.CategoryID}");
        }
    }
}
/* Output:
    Beverages
      Cola      1
      Tea       1
    Condiments
    Mustard     2
    Pickles     2
    Fruit
      Melons    5
      Peaches   5
    Grains
    Vegetables
      Bok Choy  3
      Carrots   3
*/
}

```

See also

- [Language Integrated Query \(LINQ\)](#)
- [orderby clause](#)
- [join clause](#)

Join by using composite keys

12/28/2021 • 2 minutes to read • [Edit Online](#)

This example shows how to perform join operations in which you want to use more than one key to define a match. This is accomplished by using a composite key. You create a composite key as an anonymous type or named typed with the values that you want to compare. If the query variable will be passed across method boundaries, use a named type that overrides [Equals](#) and [GetHashCode](#) for the key. The names of the properties, and the order in which they occur, must be identical in each key.

Example

The following example demonstrates how to use a composite key to join data from three tables:

```
var query = from o in db.Orders
            from p in db.Products
            join d in db.OrderDetails
              on new {o.OrderID, p.ProductID} equals new {d.OrderID, d.ProductID} into details
            from d in details
            select new {o.OrderID, p.ProductID, d.UnitPrice};
```

Type inference on composite keys depends on the names of the properties in the keys, and the order in which they occur. If the properties in the source sequences don't have the same names, you must assign new names in the keys. For example, if the `Orders` table and `OrderDetails` table each used different names for their columns, you could create composite keys by assigning identical names in the anonymous types:

```
join...on new {Name = o.CustomerName, ID = o.CustID} equals
           new {Name = d.CustName, ID = d.CustID }
```

Composite keys can be also used in a `group` clause.

See also

- [Language Integrated Query \(LINQ\)](#)
- [join clause](#)
- [group clause](#)

Perform custom join operations

12/28/2021 • 5 minutes to read • [Edit Online](#)

This example shows how to perform join operations that aren't possible with the `join` clause. In a query expression, the `join` clause is limited to, and optimized for, equijoins, which are by far the most common type of join operation. When performing an equijoin, you will probably always get the best performance by using the `join` clause.

However, the `join` clause cannot be used in the following cases:

- When the join is predicated on an expression of inequality (a non-equijoin).
- When the join is predicated on more than one expression of equality or inequality.
- When you have to introduce a temporary range variable for the right side (inner) sequence before the join operation.

To perform joins that aren't equijoins, you can use multiple `from` clauses to introduce each data source independently. You then apply a predicate expression in a `where` clause to the range variable for each source. The expression also can take the form of a method call.

NOTE

Don't confuse this kind of custom join operation with the use of multiple `from` clauses to access inner collections. For more information, see [join clause](#).

Example 1

The first method in the following example shows a simple cross join. Cross joins must be used with caution because they can produce very large result sets. However, they can be useful in some scenarios for creating source sequences against which additional queries are run.

The second method produces a sequence of all the products whose category ID is listed in the category list on the left side. Note the use of the `let` clause and the `Contains` method to create a temporary array. It also is possible to create the array before the query and eliminate the first `from` clause.

```
class CustomJoins
{
    #region Data

    class Product
    {
        public string Name { get; set; }
        public int CategoryID { get; set; }
    }

    class Category
    {
        public string Name { get; set; }
        public int ID { get; set; }
    }

    // Specify the first data source.
    List<Category> categories = new List<Category>()
```

```

{
    new Category(){Name="Beverages", ID=001},
    new Category(){ Name="Condiments", ID=002},
    new Category(){ Name="Vegetables", ID=003},
};

// Specify the second data source.
List<Product> products = new List<Product>()
{
    new Product{Name="Tea", CategoryID=001},
    new Product{Name="Mustard", CategoryID=002},
    new Product{Name="Pickles", CategoryID=002},
    new Product{Name="Carrots", CategoryID=003},
    new Product{Name="Bok Choy", CategoryID=003},
    new Product{Name="Peaches", CategoryID=005},
    new Product{Name="Melons", CategoryID=005},
    new Product{Name="Ice Cream", CategoryID=007},
    new Product{Name="Mackerel", CategoryID=012},
};
#endregion

static void Main()
{
    CustomJoins app = new CustomJoins();
    app.CrossJoin();
    app.NonEquiJoin();

    Console.WriteLine("Press any key to exit.");
    Console.ReadKey();
}

void CrossJoin()
{
    var crossJoinQuery =
        from c in categories
        from p in products
        select new { c.ID, p.Name };

    Console.WriteLine("Cross Join Query:");
    foreach (var v in crossJoinQuery)
    {
        Console.WriteLine($"{v.ID,-5}{v.Name}");
    }
}

void NonEquiJoin()
{
    var nonEquiJoinQuery =
        from p in products
        let catIds = from c in categories
                     select c.ID
        where catIds.Contains(p.CategoryID) == true
        select new { Product = p.Name, CategoryID = p.CategoryID };

    Console.WriteLine("Non-equiJoin query:");
    foreach (var v in nonEquiJoinQuery)
    {
        Console.WriteLine($"{v.CategoryID,-5}{v.Product}");
    }
}
}

/* Output:
Cross Join Query:
1 Tea
1 Mustard
1 Pickles
1 Carrots
1 Bok Choy
1 Peaches

```

```

1  Melons
1  Ice Cream
1  Mackerel
2  Tea
2  Mustard
2  Pickles
2  Carrots
2  Bok Choy
2  Peaches
2  Melons
2  Ice Cream
2  Mackerel
3  Tea
3  Mustard
3  Pickles
3  Carrots
3  Bok Choy
3  Peaches
3  Melons
3  Ice Cream
3  Mackerel
Non-equijoin query:
1  Tea
2  Mustard
2  Pickles
3  Carrots
3  Bok Choy
Press any key to exit.
*/

```

Example 2

In the following example, the query must join two sequences based on matching keys that, in the case of the inner (right side) sequence, cannot be obtained prior to the join clause itself. If this join were performed with a `join` clause, then the `Split` method would have to be called for each element. The use of multiple `from` clauses enables the query to avoid the overhead of the repeated method call. However, since `join` is optimized, in this particular case it might still be faster than using multiple `from` clauses. The results will vary depending primarily on how expensive the method call is.

```

class MergeTwoCSVFiles
{
    static void Main()
    {
        // See section Compiling the Code for information about the data files.
        string[] names = System.IO.File.ReadAllLines(@"../../names.csv");
        string[] scores = System.IO.File.ReadAllLines(@"../../scores.csv");

        // Merge the data sources using a named type.
        // You could use var instead of an explicit type for the query.
        IEnumerable<Student> queryNamesScores =
            // Split each line in the data files into an array of strings.
            from name in names
            let x = name.Split(',')
            from score in scores
            let s = score.Split(',')
            // Look for matching IDs from the two data files.
            where x[2] == s[0]
            // If the IDs match, build a Student object.
            select new Student()
            {
                FirstName = x[0],
                LastName = x[1],
                ID = Convert.ToInt32(x[2]),
                ExamScores = (from scoreAsText in s.Skip(1)

```

```

        select Convert.ToInt32(scoreAsText)).
        ToList()

    };

    // Optional. Store the newly created student objects in memory
    // for faster access in future queries
    List<Student> students = queryNamesScores.ToList();

    foreach (var student in students)
    {
        Console.WriteLine($"The average score of {student.FirstName} {student.LastName} is
{student.ExamScores.Average()}.");
    }

    //Keep console window open in debug mode
    Console.WriteLine("Press any key to exit.");
    Console.ReadKey();
}
}

class Student
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public int ID { get; set; }
    public List<int> ExamScores { get; set; }
}

/* Output:
    The average score of Omelchenko Svetlana is 82.5.
    The average score of O'Donnell Claire is 72.25.
    The average score of Mortensen Sven is 84.5.
    The average score of Garcia Cesar is 88.25.
    The average score of Garcia Debra is 67.
    The average score of Fakhouri Fadi is 92.25.
    The average score of Feng Hanying is 88.
    The average score of Garcia Hugo is 85.75.
    The average score of Tucker Lance is 81.75.
    The average score of Adams Terry is 85.25.
    The average score of Zabokritski Eugene is 83.
    The average score of Tucker Michael is 92.
*/

```

See also

- [Language Integrated Query \(LINQ\)](#)
- [join clause](#)
- [Order the results of a join clause](#)

Handle null values in query expressions

12/28/2021 • 2 minutes to read • [Edit Online](#)

This example shows how to handle possible null values in source collections. An object collection such as an [IEnumerable<T>](#) can contain elements whose value is [null](#). If a source collection is `null` or contains an element whose value is `null`, and your query doesn't handle `null` values, a [NullReferenceException](#) will be thrown when you execute the query.

You can code defensively to avoid a null reference exception as shown in the following example:

```
var query1 =
    from c in categories
    where c != null
    join p in products on c.ID equals
        p?.CategoryID
    select new { Category = c.Name, Name = p.Name };
```

In the previous example, the `where` clause filters out all null elements in the categories sequence. This technique is independent of the null check in the join clause. The conditional expression with null in this example works because `Products.CategoryID` is of type `int?`, which is shorthand for `Nullable<int>`.

In a join clause, if only one of the comparison keys is a nullable value type, you can cast the other to a nullable value type in the query expression. In the following example, assume that `EmployeeID` is a column that contains values of type `int?`:

```
void TestMethod(Northwind db)
{
    var query =
        from o in db.Orders
        join e in db.Employees
            on o.EmployeeID equals (int?)e.EmployeeID
        select new { o.OrderID, e.FirstName };
}
```

In each of the examples, the `equals` query keyword is used. C# 9 adds [pattern matching](#), which includes patterns for `is null` and `is not null`. These patterns aren't recommended in LINQ queries because query providers may not interpret the new C# syntax correctly. A query provider is a library that translates C# query expressions into a native data format, such as Entity Framework Core. Query providers implement the [System.Linq.IQueryProvider](#) interface to create data sources that implement the [System.Linq.IQueryable<T>](#) interface.

See also

- [Nullable<T>](#)
- [Language Integrated Query \(LINQ\)](#)
- [Nullable value types](#)

Handle exceptions in query expressions

12/28/2021 • 2 minutes to read • [Edit Online](#)

It's possible to call any method in the context of a query expression. However, we recommend that you avoid calling any method in a query expression that can create a side effect such as modifying the contents of the data source or throwing an exception. This example shows how to avoid raising exceptions when you call methods in a query expression without violating the general .NET guidelines on exception handling. Those guidelines state that it's acceptable to catch a specific exception when you understand why it's thrown in a given context. For more information, see [Best Practices for Exceptions](#).

The final example shows how to handle those cases when you must throw an exception during execution of a query.

Example 1

The following example shows how to move exception handling code outside a query expression. This is only possible when the method does not depend on any variables local to the query.

```
class ExceptionsOutsideQuery
{
    static void Main()
    {
        // DO THIS with a datasource that might
        // throw an exception. It is easier to deal with
        // outside of the query expression.
        IEnumerable<int> dataSource;
        try
        {
            dataSource = GetData();
        }
        catch (InvalidOperationException)
        {
            // Handle (or don't handle) the exception
            // in the way that is appropriate for your application.
            Console.WriteLine("Invalid operation");
            goto Exit;
        }

        // If we get here, it is safe to proceed.
        var query = from i in dataSource
                    select i * i;

        foreach (var i in query)
            Console.WriteLine(i.ToString());

        //Keep the console window open in debug mode
        Exit:
        Console.WriteLine("Press any key to exit");
        Console.ReadKey();
    }

    // A data source that is very likely to throw an exception!
    static IEnumerable<int> GetData()
    {
        throw new InvalidOperationException();
    }
}
```


Example 2

In some cases, the best response to an exception that is thrown from within a query might be to stop the query execution immediately. The following example shows how to handle exceptions that might be thrown from inside a query body. Assume that `SomeMethodThatMightThrow` can potentially cause an exception that requires the query execution to stop.

Note that the `try` block encloses the `foreach` loop, and not the query itself. This is because the `foreach` loop is the point at which the query is actually executed. For more information, see [Introduction to LINQ queries](#).

```
class QueryThatThrows
{
    static void Main()
    {
        // Data source.
        string[] files = { "fileA.txt", "fileB.txt", "fileC.txt" };

        // Demonstration query that throws.
        var exceptionDemoQuery =
            from file in files
            let n = SomeMethodThatMightThrow(file)
            select n;

        // Runtime exceptions are thrown when query is executed.
        // Therefore they must be handled in the foreach loop.
        try
        {
            foreach (var item in exceptionDemoQuery)
            {
                Console.WriteLine($"Processing {item}");
            }
        }

        // Catch whatever exception you expect to raise
        // and/or do any necessary cleanup in a finally block
        catch (InvalidOperationException e)
        {
            Console.WriteLine(e.Message);
        }

        //Keep the console window open in debug mode
        Console.WriteLine("Press any key to exit");
        Console.ReadKey();
    }

    // Not very useful as a general purpose method.
    static string SomeMethodThatMightThrow(string s)
    {
        if (s[4] == 'C')
            throw new InvalidOperationException();
        return @"C:\newFolder\" + s;
    }
}

/* Output:
Processing C:\newFolder\fileA.txt
Processing C:\newFolder\fileB.txt
Operation is not valid due to the current state of the object.
*/
```

See also

- [Language Integrated Query \(LINQ\)](#)

Write safe and efficient C# code

12/28/2021 • 17 minutes to read • [Edit Online](#)

C# provides features that enable you to write verifiable safe code with better performance. If you carefully apply these techniques, fewer scenarios require unsafe code. These features make it easier to use references to value types as method arguments and method returns. When done safely, these techniques minimize copying value types. By using value types, you can minimize the number of allocations and garbage collection passes.

Much of the sample code in this article uses features added in C# 7.2. To use those features, make sure your project isn't configured to use an earlier version. For more information, see [configure the language version](#).

One advantage to using value types is that they often avoid heap allocations. The disadvantage is that they're copied by value. This trade-off makes it harder to optimize algorithms that operate on large amounts of data. The language features highlighted in this article provide mechanisms that enable safe efficient code using references to value types. Use these features wisely to minimize both allocations and copy operations.

Some of the guidance in this article refers to coding practices that are always advisable, not only for the performance benefit. Use the `readonly` keyword when it accurately expresses design intent:

- Declare immutable structs as `readonly`.
- Declare `readonly` members for mutable structs.

The article also explains some low-level optimizations that are advisable when you've run a profiler and have identified bottlenecks:

- Use the `in` parameter modifier.
- Use `ref readonly return` statements.
- Use `ref struct` types.
- Use `nint` and `nuint` types.

These techniques balance two competing goals:

- Minimize allocations on the heap.

Variables that are [reference types](#) hold a reference to a location in memory and are allocated on the managed heap. Only the reference is copied when a reference type is passed as an argument to a method or returned from a method. Each new object requires a new allocation, and later must be reclaimed. Garbage collection takes time.

- Minimize the copying of values.

Variables that are [value types](#) directly contain their value, and the value is typically copied when passed to a method or returned from a method. This behavior includes copying the value of `this` when calling iterators and async instance methods of structs. The copy operation takes time, depending on the size of the type.

This article uses the following example concept of the 3D-point structure to explain its recommendations:

```
public struct Point3D
{
    public double X;
    public double Y;
    public double Z;
}
```

Different examples use different implementations of this concept.

Declare immutable structs as `readonly`

Declare a `readonly struct` to indicate that a type is **immutable**. The `readonly` modifier informs the compiler that your intent is to create an immutable type. The compiler enforces that design decision with the following rules:

- All field members must be read-only.
- All properties must be read-only, including auto-implemented properties.

These two rules are sufficient to ensure that no member of a `readonly struct` modifies the state of that struct. The `struct` is immutable. The `Point3D` structure could be defined as an immutable struct as shown in the following example:

```
readonly public struct ReadonlyPoint3D
{
    public ReadonlyPoint3D(double x, double y, double z)
    {
        this.X = x;
        this.Y = y;
        this.Z = z;
    }

    public double X { get; }
    public double Y { get; }
    public double Z { get; }
}
```

Follow this recommendation whenever your design intent is to create an immutable value type. Any performance improvements are an added benefit. The `readonly struct` keywords clearly express your design intent.

Declare `readonly` members for mutable structs

In C# 8.0 and later, when a struct type is mutable, declare members that don't modify state as `readonly` members.

Consider a different application that needs a 3D point structure, but must support mutability. The following version of the 3D point structure adds the `readonly` modifier only to those members that don't modify the structure. Follow this example when your design must support modifications to the struct by some members, but you still want the benefits of enforcing `readonly` on some members:

```

public struct Point3D
{
    public Point3D(double x, double y, double z)
    {
        _x = x;
        _y = y;
        _z = z;
    }

    private double _x;
    public double X
    {
        readonly get => _x;
        set => _x = value;
    }

    private double _y;
    public double Y
    {
        readonly get => _y;
        set => _y = value;
    }

    private double _z;
    public double Z
    {
        readonly get => _z;
        set => _z = value;
    }

    public readonly double Distance => Math.Sqrt(X * X + Y * Y + Z * Z);

    public readonly override string ToString() => $"{X}, {Y}, {Z}";
}

```

The preceding sample shows many of the locations where you can apply the `readonly` modifier: methods, properties, and property accessors. If you use auto-implemented properties, the compiler adds the `readonly` modifier to the `get` accessor for read-write properties. The compiler adds the `readonly` modifier to the auto-implemented property declarations for properties with only a `get` accessor.

Adding the `readonly` modifier to members that don't mutate state provides two related benefits. First, the compiler enforces your intent. That member can't mutate the struct's state. Second, the compiler won't create [defensive copies](#) of `in` parameters when accessing a `readonly` member. The compiler can make this optimization safely because it guarantees that the `struct` is not modified by a `readonly` member.

Use `ref readonly return` statements

Use a `ref readonly` return when both of the following conditions are true:

- The return value is a `struct` larger than `IntPtr.Size`.
- The storage lifetime is greater than the method returning the value.

You can return values by reference when the value being returned isn't local to the returning method. Returning by reference means that only the reference is copied, not the structure. In the following example, the `Origin` property can't use a `ref` return because the value being returned is a local variable:

```

public Point3D Origin => new Point3D(0,0,0);

```

However, the following property definition can be returned by reference because the returned value is a static

member:

```
public struct Point3D
{
    private static Point3D origin = new Point3D(0,0,0);

    // Dangerous! returning a mutable reference to internal storage
    public ref Point3D Origin => ref origin;

    // other members removed for space
}
```

You don't want callers modifying the origin, so you should return the value by `ref readonly`:

```
public struct Point3D
{
    private static Point3D origin = new Point3D(0,0,0);

    public static ref readonly Point3D Origin => ref origin;

    // other members removed for space
}
```

Returning `ref readonly` enables you to save copying larger structures and preserve the immutability of your internal data members.

At the call site, callers make the choice to use the `Origin` property as a `ref readonly` or as a value:

```
var originValue = Point3D.Origin;
ref readonly var originReference = ref Point3D.Origin;
```

The first assignment in the preceding code makes a copy of the `Origin` constant and assigns that copy. The second assigns a reference. Notice that the `readonly` modifier must be part of the declaration of the variable. The reference to which it refers can't be modified. Attempts to do so result in a compile-time error.

The `readonly` modifier is required on the declaration of `originReference`.

The compiler enforces that the caller can't modify the reference. Attempts to assign the value directly generate a compile-time error. In other cases, the compiler allocates a [defensive copy](#) unless it can safely use the `readonly` reference. Static analysis rules determine if the struct could be modified. The compiler doesn't create a defensive copy when the struct is a `readonly struct` or the member is a `readonly` member of the struct. Defensive copies aren't needed to pass the struct as an `in` argument.

Use the `in` parameter modifier

The following sections explain what the `in` modifier does, how to use it, and when to use it for performance optimization:

- [The `out`, `ref`, and `in` keywords](#)
- [Use `in` parameters for large structs](#)
- [Optional use of `in` at call site](#)
- [Avoid defensive copies](#)

The `out`, `ref`, and `in` keywords

The `in` keyword complements the `ref` and `out` keywords to pass arguments by reference. The `in` keyword

specifies that the argument is passed by reference, but the called method doesn't modify the value. The `in` modifier can be applied to any member that takes parameters, such as methods, delegates, lambdas, local functions, indexers, and operators.

With the addition of the `in` keyword, C# provides a full vocabulary to express your design intent. Value types are copied when passed to a called method when you don't specify any of the following modifiers in the method signature. Each of these modifiers specifies that a variable is passed by reference, avoiding the copy. Each modifier expresses a different intent:

- `out`: This method sets the value of the argument used as this parameter.
- `ref`: This method may modify the value of the argument used as this parameter.
- `in`: This method doesn't modify the value of the argument used as this parameter.

Add the `in` modifier to pass an argument by reference and declare your design intent to pass arguments by reference to avoid unnecessary copying. You don't intend to modify the object used as that argument.

The `in` modifier complements `out` and `ref` in other ways as well. You can't create overloads of a method that differ only in the presence of `in`, `out`, or `ref`. These new rules extend the same behavior that had always been defined for `out` and `ref` parameters. Like the `out` and `ref` modifiers, value types aren't boxed because the `in` modifier is applied. Another feature of `in` parameters is that you can use literal values or constants for the argument to an `in` parameter.

The `in` modifier can also be used with reference types or numeric values. However, the benefits in those cases are minimal, if any.

There are several ways in which the compiler enforces the read-only nature of an `in` argument. First of all, the called method can't directly assign to an `in` parameter. It can't directly assign to any field of an `in` parameter when that value is a `struct` type. In addition, you can't pass an `in` parameter to any method using the `ref` or `out` modifier. These rules apply to any field of an `in` parameter, provided the field is a `struct` type and the parameter is also a `struct` type. In fact, these rules apply for multiple layers of member access provided the types at all levels of member access are `structs`. The compiler enforces that `struct` types passed as `in` arguments and their `struct` members are read-only variables when used as arguments to other methods.

Use `in` parameters for large structs

You can apply the `in` modifier to any `readonly struct` parameter, but this practice is likely to improve performance only for value types that are substantially larger than `IntPtr.Size`. For simple types (such as `sbyte`, `byte`, `short`, `ushort`, `int`, `uint`, `long`, `ulong`, `char`, `float`, `double`, `decimal` and `bool`, and `enum` types), any potential performance gains are minimal. Some simple types, such as `decimal` at 16 bytes in size, are larger than either 4-byte or 8-byte references but not by enough to make a measurable difference in performance in most scenarios. And performance may degrade by using pass-by-reference for types smaller than `IntPtr.Size`.

The following code shows an example of a method that calculates the distance between two points in 3D space.

```
private static double CalculateDistance(in Point3D point1, in Point3D point2)
{
    double xDifference = point1.X - point2.X;
    double yDifference = point1.Y - point2.Y;
    double zDifference = point1.Z - point2.Z;

    return Math.Sqrt(xDifference * xDifference + yDifference * yDifference + zDifference * zDifference);
}
```

The arguments are two structures that each contain three doubles. A double is 8 bytes, so each argument is 24 bytes. By specifying the `in` modifier, you pass a 4-byte or 8-byte reference to those arguments, depending on the architecture of the machine. The difference in size is small, but it can add up when your application calls this

method in a tight loop using many different values.

However, the impact of any low-level optimizations like using the `in` modifier should be measured to validate a performance benefit. For example, you might think that using `in` on a `Guid` parameter would be beneficial. The `Guid` type is 16 bytes in size, twice the size of an 8-byte reference. But such a small difference isn't likely to result in a measurable performance benefit unless it's in a method that's in a time critical hot path for your application.

Optional use of `in` at call site

Unlike a `ref` or `out` parameter, you don't need to apply the `in` modifier at the call site. The following code shows two examples of calling the `CalculateDistance` method. The first uses two local variables passed by reference. The second includes a temporary variable created as part of the method call.

```
var distance = CalculateDistance(pt1, pt2);
var fromOrigin = CalculateDistance(pt1, new Point3D());
```

Omitting the `in` modifier at the call site informs the compiler that it's allowed to make a copy of the argument for any of the following reasons:

- There exists an implicit conversion but not an identity conversion from the argument type to the parameter type.
- The argument is an expression but doesn't have a known storage variable.
- An overload exists that differs by the presence or absence of `in`. In that case, the by value overload is a better match.

These rules are useful as you update existing code to use read-only reference arguments. Inside the called method, you can call any instance method that uses by-value parameters. In those instances, a copy of the `in` parameter is created.

Because the compiler may create a temporary variable for any `in` parameter, you can also specify default values for any `in` parameter. The following code specifies the origin (point 0,0,0) as the default value for the second point:

```
private static double CalculateDistance2(in Point3D point1, in Point3D point2 = default)
{
    double xDifference = point1.X - point2.X;
    double yDifference = point1.Y - point2.Y;
    double zDifference = point1.Z - point2.Z;

    return Math.Sqrt(xDifference * xDifference + yDifference * yDifference + zDifference * zDifference);
}
```

To force the compiler to pass read-only arguments by reference, specify the `in` modifier on the arguments at the call site, as shown in the following code:

```
distance = CalculateDistance(in pt1, in pt2);
distance = CalculateDistance(in pt1, new Point3D());
distance = CalculateDistance(pt1, in Point3D.Origin);
```

This behavior makes it easier to adopt `in` parameters over time in large codebases where performance gains are possible. You add the `in` modifier to method signatures first. Then you can add the `in` modifier at call sites and create `readonly struct` types to enable the compiler to avoid creating defensive copies of `in` parameters in more locations.

Avoid defensive copies

Pass a `struct` as the argument for an `in` parameter only if it's declared with the `readonly` modifier or the method accesses only `readonly` members of the struct. Otherwise, the compiler must create *defensive copies* in many situations to ensure that arguments are not mutated. Consider the following example that calculates the distance of a 3D point from the origin:

```
private static double CalculateDistance(in Point3D point1, in Point3D point2)
{
    double xDifference = point1.X - point2.X;
    double yDifference = point1.Y - point2.Y;
    double zDifference = point1.Z - point2.Z;

    return Math.Sqrt(xDifference * xDifference + yDifference * yDifference + zDifference * zDifference);
}
```

The `Point3D` structure is *not* a read-only struct. There are six different property access calls in the body of this method. On first examination, you may think these accesses are safe. After all, a `get` accessor shouldn't modify the state of the object. But there's no language rule that enforces that. It's only a common convention. Any type could implement a `get` accessor that modified the internal state.

Without some language guarantee, the compiler must create a temporary copy of the argument before calling any member not marked with the `readonly` modifier. The temporary storage is created on the stack, the values of the argument are copied to the temporary storage, and the value is copied to the stack for each member access as the `this` argument. In many situations, these copies harm performance enough that pass-by-value is faster than pass-by-read-only-reference when the argument type isn't a `readonly struct` and the method calls members that aren't marked `readonly`. If you mark all methods that don't modify the struct state as `readonly`, the compiler can safely determine that the struct state isn't modified, and a defensive copy is not needed.

If the distance calculation uses the immutable struct, `ReadOnlyPoint3D`, temporary objects aren't needed:

```
private static double CalculateDistance3(in ReadOnlyPoint3D point1, in ReadOnlyPoint3D point2 = default)
{
    double xDifference = point1.X - point2.X;
    double yDifference = point1.Y - point2.Y;
    double zDifference = point1.Z - point2.Z;

    return Math.Sqrt(xDifference * xDifference + yDifference * yDifference + zDifference * zDifference);
}
```

The compiler generates more efficient code when you call members of a `readonly struct`. The `this` reference, instead of a copy of the receiver, is always an `in` parameter passed by reference to the member method. This optimization saves copying when you use a `readonly struct` as an `in` argument.

Don't pass a nullable value type as an `in` argument. The `Nullable<T>` type isn't declared as a read-only struct. That means the compiler must generate defensive copies for any nullable value type argument passed to a method using the `in` modifier on the parameter declaration.

You can see an example program that demonstrates the performance differences using [BenchmarkDotNet](#) in our [samples repository](#) on GitHub. It compares passing a mutable struct by value and by reference with passing an immutable struct by value and by reference. The use of the immutable struct and pass by reference is fastest.

Use `ref struct` types

Use a `ref struct` or a `readonly ref struct`, such as `Span<T>` or `ReadOnlySpan<T>`, to work with blocks of memory as a sequence of bytes. The memory used by the span is constrained to a single stack frame. This restriction enables the compiler to make several optimizations. The primary motivation for this feature was `Span<T>` and related structures. You'll achieve performance improvements from these enhancements by using

new and updated .NET APIs that make use of the [Span<T>](#) type.

Declaring a struct as `readonly ref` combines the benefits and restrictions of `ref struct` and `readonly struct` declarations. The memory used by the readonly span is restricted to a single stack frame, and the memory used by the readonly span can't be modified.

You may have similar requirements working with memory created using `stackalloc` or when using memory from interop APIs. You can define your own `ref struct` types for those needs.

Use `nint` and `nuint` types

[Native-sized integer types](#) are 32-bit integers in a 32-bit process or 64-bit integers in a 64-bit process. Use them for interop scenarios, low-level libraries, and to optimize performance in scenarios where integer math is used extensively.

Conclusions

Using value types minimizes the number of allocation operations:

- Storage for value types is stack-allocated for local variables and method arguments.
- Storage for value types that are members of other objects is allocated as part of that object, not as a separate allocation.
- Storage for value type return values is stack allocated.

Contrast that with reference types in those same situations:

- Storage for reference types is heap allocated for local variables and method arguments. The reference is stored on the stack.
- Storage for reference types that are members of other objects are separately allocated on the heap. The containing object stores the reference.
- Storage for reference type return values is heap allocated. The reference to that storage is stored on the stack.

Minimizing allocations comes with tradeoffs. You copy more memory when the size of the `struct` is larger than the size of a reference. A reference is typically 64 bits or 32 bits, and depends on the target machine CPU.

These tradeoffs generally have minimal performance impact. However, for large structs or larger collections, the performance impact increases. The impact can be large in tight loops and hot paths for programs.

These enhancements to the C# language are designed for performance critical algorithms where minimizing memory allocations is a major factor in achieving the necessary performance. You may find that you don't often use these features in the code you write. However, these enhancements have been adopted throughout .NET. As more APIs make use of these features, you'll see the performance of your applications improve.

See also

- [in parameter modifier \(C# Reference\)](#)
- [ref keyword](#)
- [Ref returns and ref locals](#)

Expression Trees

12/28/2021 • 2 minutes to read • [Edit Online](#)

If you have used LINQ, you have experience with a rich library where the `Func` types are part of the API set. (If you are not familiar with LINQ, you probably want to read [the LINQ tutorial](#) and the article about [lambda expressions](#) before this one.) *Expression Trees* provide richer interaction with the arguments that are functions.

You write function arguments, typically using Lambda Expressions, when you create LINQ queries. In a typical LINQ query, those function arguments are transformed into a delegate the compiler creates.

When you want to have a richer interaction, you need to use *Expression Trees*. Expression Trees represent code as a structure that you can examine, modify, or execute. These tools give you the power to manipulate code during run time. You can write code that examines running algorithms, or injects new capabilities. In more advanced scenarios, you can modify running algorithms, and even translate C# expressions into another form for execution in another environment.

You've likely already written code that uses Expression Trees. Entity Framework's LINQ APIs accept Expression Trees as the arguments for the LINQ Query Expression Pattern. That enables [Entity Framework](#) to translate the query you wrote in C# into SQL that executes in the database engine. Another example is [Moq](#), which is a popular mocking framework for .NET.

The remaining sections of this tutorial will explore what Expression Trees are, examine the framework classes that support expression trees, and show you how to work with expression trees. You'll learn how to read expression trees, how to create expression trees, how to create modified expression trees, and how to execute the code represented by expression trees. After reading, you will be ready to use these structures to create rich adaptive algorithms.

1. [Expression Trees Explained](#)

Understand the structure and concepts behind *Expression Trees*.

2. [Framework Types Supporting Expression Trees](#)

Learn about the structures and classes that define and manipulate expression trees.

3. [Executing Expressions](#)

Learn how to convert an expression tree represented as a Lambda Expression into a delegate and execute the resulting delegate.

4. [Interpreting Expressions](#)

Learn how to traverse and examine *expression trees* to understand what code the expression tree represents.

5. [Building Expressions](#)

Learn how to construct the nodes for an expression tree and build expression trees.

6. [Translating Expressions](#)

Learn how to build a modified copy of an expression tree, or translate an expression tree into a different format.

7. [Summing up](#)

Review the information on expression trees.

Expression Trees Explained

12/28/2021 • 4 minutes to read • [Edit Online](#)

[Previous](#) -- [Overview](#)

An Expression Tree is a data structure that defines code. They are based on the same structures that a compiler uses to analyze code and generate the compiled output. As you read through this tutorial, you will notice quite a bit of similarity between Expression Trees and the types used in the Roslyn APIs to build [Analyzers and CodeFixes](#). (Analyzers and CodeFixes are NuGet packages that perform static analysis on code and can suggest potential fixes for a developer.) The concepts are similar, and the end result is a data structure that allows examination of the source code in a meaningful way. However, Expression Trees are based on a totally different set of classes and APIs than the Roslyn APIs.

Let's look at a simple example. Here's a line of code:

```
var sum = 1 + 2;
```

If you were to analyze this as an expression tree, the tree contains several nodes. The outermost node is a variable declaration statement with assignment (`var sum = 1 + 2;`) That outermost node contains several child nodes: a variable declaration, an assignment operator, and an expression representing the right hand side of the equals sign. That expression is further subdivided into expressions that represent the addition operation, and left and right operands of the addition.

Let's drill down a bit more into the expressions that make up the right side of the equals sign. The expression is `1 + 2`. That's a binary expression. More specifically, it's a binary addition expression. A binary addition expression has two children, representing the left and right nodes of the addition expression. Here, both nodes are constant expressions: The left operand is the value `1`, and the right operand is the value `2`.

Visually, the entire statement is a tree: You could start at the root node, and travel to each node in the tree to see the code that makes up the statement:

- Variable declaration statement with assignment (`var sum = 1 + 2;`)
 - Implicit variable type declaration (`var sum`)
 - Implicit var keyword (`var`)
 - Variable name declaration (`sum`)
 - Assignment operator (`=`)
 - Binary addition expression (`1 + 2`)
 - Left operand (`1`)
 - Addition operator (`+`)
 - Right operand (`2`)

This may look complicated, but it is very powerful. Following the same process, you can decompose much more complicated expressions. Consider this expression:

```
var finalAnswer = this.SecretSauceFunction(  
    currentState.createInterimResult(), currentState.createSecondValue(1, 2),  
    decisionServer.considerFinalOptions("hello")) +  
    MoreSecretSauce('A', DateTime.Now, true);
```

The expression above is also a variable declaration with an assignment. In this instance, the right hand side of the assignment is a much more complicated tree. I'm not going to decompose this expression, but consider what the different nodes might be. There are method calls using the current object as a receiver, one that has an explicit `this` receiver, one that does not. There are method calls using other receiver objects, there are constant arguments of different types. And finally, there is a binary addition operator. Depending on the return type of `SecretSauceFunction()` or `MoreSecretSauce()`, that binary addition operator may be a method call to an overridden addition operator, resolving to a static method call to the binary addition operator defined for a class.

Despite this perceived complexity, the expression above creates a tree structure that can be navigated as easily as the first sample. You can keep traversing child nodes to find leaf nodes in the expression. Parent nodes will have references to their children, and each node has a property that describes what kind of node it is.

The structure of an expression tree is very consistent. Once you've learned the basics, you can understand even the most complex code when it is represented as an expression tree. The elegance in the data structure explains how the C# compiler can analyze the most complex C# programs and create proper output from that complicated source code.

Once you become familiar with the structure of expression trees, you will find that knowledge you've gained quickly enables you to work with many more and more advanced scenarios. There is incredible power to expression trees.

In addition to translating algorithms to execute in other environments, expression trees can be used to make it easier to write algorithms that inspect code before executing it. You can write a method whose arguments are expressions and then examine those expressions before executing the code. The Expression Tree is a full representation of the code: you can see values of any sub-expression. You can see method and property names. You can see the value of any constant expressions. You can also convert an expression tree into an executable delegate, and execute the code.

The APIs for Expression Trees enable you to create trees that represent almost any valid code construct. However, to keep things as simple as possible, some C# idioms cannot be created in an expression tree. One example is asynchronous expressions (using the `async` and `await` keywords). If your needs require asynchronous algorithms, you would need to manipulate the `Task` objects directly, rather than rely on the compiler support. Another is in creating loops. Typically, you create these by using `for`, `foreach`, `while` or `do` loops. As you'll see [later in this series](#), the APIs for expression trees support a single loop expression, with `break` and `continue` expressions that control repeating the loop.

The one thing you can't do is modify an expression tree. Expression Trees are immutable data structures. If you want to mutate (change) an expression tree, you must create a new tree that is a copy of the original, but with your desired changes.

[Next -- Framework Types Supporting Expression Trees](#)

Framework Types Supporting Expression Trees

12/28/2021 • 3 minutes to read • [Edit Online](#)

[Previous -- Expression Trees Explained](#)

There is a large list of classes in the .NET Core framework that work with Expression Trees. You can see the full list at [System.Linq.Expressions](#). Rather than run through the full list, let's understand how the framework classes have been designed.

In language design, an expression is a body of code that evaluates and returns a value. Expressions may be very simple: the constant expression `1` returns the constant value of 1. They may be more complicated: The expression `(-B + Math.Sqrt(B*B - 4 * A * C)) / (2 * A)` returns one root for a quadratic equation (in the case where the equation has a solution).

It all starts with System.Linq.Expression

One of the complexities of working with expression trees is that many different kinds of expressions are valid in many places in programs. Consider an assignment expression. The right hand side of an assignment could be a constant value, a variable, a method call expression, or others. That language flexibility means that you may encounter many different expression types anywhere in the nodes of a tree when you traverse an expression tree. Therefore, when you can work with the base expression type, that's the simplest way to work. However, sometimes you need to know more. The base Expression class contains a `NodeType` property for this purpose. It returns an `ExpressionType` which is an enumeration of possible expression types. Once you know the type of the node, you can cast it to that type, and perform specific actions knowing the type of the expression node. You can search for certain node types, and then work with the specific properties of that kind of expression.

For example, this code will print the name of a variable for a variable access expression. I've followed the practice of checking the node type, then casting to a variable access expression and then checking the properties of the specific expression type:

```
Expression<Func<int, int>> addFive = (num) => num + 5;

if (addFive.NodeType == ExpressionType.Lambda)
{
    var lambdaExp = (LambdaExpression)addFive;

    var parameter = lambdaExp.Parameters.First();

    Console.WriteLine(parameter.Name);
    Console.WriteLine(parameter.Type);
}
```

Creating Expression Trees

The `System.Linq.Expression` class also contains many static methods to create expressions. These methods create an expression node using the arguments supplied for its children. In this way, you build an expression up from its leaf nodes. For example, this code builds an Add expression:

```
// Addition is an add expression for "1 + 2"
var one = Expression.Constant(1, typeof(int));
var two = Expression.Constant(2, typeof(int));
var addition = Expression.Add(one, two);
```

You can see from this simple example that many types are involved in creating and working with expression trees. That complexity is necessary to provide the capabilities of the rich vocabulary provided by the C# language.

Navigating the APIs

There are Expression node types that map to almost all of the syntax elements of the C# language. Each type has specific methods for that type of language element. It's a lot to keep in your head at one time. Rather than try to memorize everything, here are the techniques I use to work with Expression trees:

1. Look at the members of the `ExpressionType` enum to determine possible nodes you should be examining. This really helps when you want to traverse and understand an expression tree.
2. Look at the static members of the `Expression` class to build an expression. Those methods can build any expression type from a set of its child nodes.
3. Look at the `ExpressionVisitor` class to build a modified expression tree.

You'll find more as you look at each of those three areas. Invariably, you will find what you need when you start with one of those three steps.

[Next -- Executing Expression Trees](#)

Executing Expression Trees

12/28/2021 • 6 minutes to read • [Edit Online](#)

[Previous -- Framework Types Supporting Expression Trees](#)

An *expression tree* is a data structure that represents some code. It is not compiled and executable code. If you want to execute the .NET code that is represented by an expression tree, you must convert it into executable IL instructions.

Lambda Expressions to Functions

You can convert any `LambdaExpression`, or any type derived from `LambdaExpression` into executable IL. Other expression types cannot be directly converted into code. This restriction has little effect in practice. Lambda expressions are the only types of expressions that you would want to execute by converting to executable intermediate language (IL). (Think about what it would mean to directly execute a `ConstantExpression`. Would it mean anything useful?) Any expression tree that is a `LambdaExpression`, or a type derived from `LambdaExpression` can be converted to IL. The expression type `Expression<TDelegate>` is the only concrete example in the .NET Core libraries. It's used to represent an expression that maps to any delegate type. Because this type maps to a delegate type, .NET can examine the expression, and generate IL for an appropriate delegate that matches the signature of the lambda expression.

In most cases, this creates a simple mapping between an expression, and its corresponding delegate. For example, an expression tree that is represented by `Expression<Func<int>>` would be converted to a delegate of the type `Func<int>`. For a lambda expression with any return type and argument list, there exists a delegate type that is the target type for the executable code represented by that lambda expression.

The `LambdaExpression` type contains `Compile` and `CompileToMethod` members that you would use to convert an expression tree to executable code. The `Compile` method creates a delegate. The `CompileToMethod` method updates a `MethodBuilder` object with the IL that represents the compiled output of the expression tree. Note that `CompileToMethod` is only available in the full desktop framework, not in the .NET Core.

Optionally, you can also provide a `DebugInfoGenerator` that will receive the symbol debugging information for the generated delegate object. This enables you to convert the expression tree into a delegate object, and have full debugging information about the generated delegate.

You would convert an expression into a delegate using the following code:

```
Expression<Func<int>> add = () => 1 + 2;
var func = add.Compile(); // Create Delegate
var answer = func(); // Invoke Delegate
Console.WriteLine(answer);
```

Notice that the delegate type is based on the expression type. You must know the return type and the argument list if you want to use the delegate object in a strongly typed manner. The `LambdaExpression.Compile()` method returns the `Delegate` type. You will have to cast it to the correct delegate type to have any compile-time tools check the argument list or return type.

Execution and Lifetimes

You execute the code by invoking the delegate created when you called `LambdaExpression.Compile()`. You can see this above where `add.Compile()` returns a delegate. Invoking that delegate, by calling `func()` executes the code.

That delegate represents the code in the expression tree. You can retain the handle to that delegate and invoke it later. You don't need to compile the expression tree each time you want to execute the code it represents. (Remember that expression trees are immutable, and compiling the same expression tree later will create a delegate that executes the same code.)

I will caution you against trying to create any more sophisticated caching mechanisms to increase performance by avoiding unnecessary compile calls. Comparing two arbitrary expression trees to determine if they represent the same algorithm will also be time consuming to execute. You'll likely find that the compute time you save avoiding any extra calls to `LambdaExpression.Compile()` will be more than consumed by the time executing code that determines if two different expression trees result in the same executable code.

Caveats

Compiling a lambda expression to a delegate and invoking that delegate is one of the simplest operations you can perform with an expression tree. However, even with this simple operation, there are caveats you must be aware of.

Lambda Expressions create closures over any local variables that are referenced in the expression. You must guarantee that any variables that would be part of the delegate are usable at the location where you call `Compile`, and when you execute the resulting delegate.

In general, the compiler will ensure that this is true. However, if your expression accesses a variable that implements `IDisposable`, it's possible that your code might dispose of the object while it is still held by the expression tree.

For example, this code works fine, because `int` does not implement `IDisposable`:

```
private static Func<int, int> CreateBoundFunc()
{
    var constant = 5; // constant is captured by the expression tree
    Expression<Func<int, int>> expression = (b) => constant + b;
    var rVal = expression.Compile();
    return rVal;
}
```

The delegate has captured a reference to the local variable `constant`. That variable is accessed at any time later, when the function returned by `CreateBoundFunc` executes.

However, consider this (rather contrived) class that implements `IDisposable`:

```
public class Resource : IDisposable
{
    private bool isDisposed = false;
    public int Argument
    {
        get
        {
            if (!isDisposed)
                return 5;
            else throw new ObjectDisposedException("Resource");
        }
    }

    public void Dispose()
    {
        isDisposed = true;
    }
}
```

If you use it in an expression as shown below, you'll get an `ObjectDisposedException` when you execute the code referenced by the `Resource.Argument` property:

```
private static Func<int, int> CreateBoundResource()
{
    using (var constant = new Resource()) // constant is captured by the expression tree
    {
        Expression<Func<int, int>> expression = (b) => constant.Argument + b;
        var rVal = expression.Compile();
        return rVal;
    }
}
```

The delegate returned from this method has closed over the `constant` object, which has been disposed of. (It's been disposed, because it was declared in a `using` statement.)

Now, when you execute the delegate returned from this method, you'll have an `ObjectDisposedException` thrown at the point of execution.

It does seem strange to have a runtime error representing a compile-time construct, but that's the world we enter when we work with expression trees.

There are a lot of permutations of this problem, so it's hard to offer general guidance to avoid it. Be careful about accessing local variables when defining expressions, and be careful about accessing state in the current object (represented by `this`) when creating an expression tree that can be returned by a public API.

The code in your expression may reference methods or properties in other assemblies. That assembly must be accessible when the expression is defined, and when it is compiled, and when the resulting delegate is invoked. You'll be met with a `ReferencedAssemblyNotFoundException` in cases where it is not present.

Summary

Expression Trees that represent lambda expressions can be compiled to create a delegate that you can execute. This provides one mechanism to execute the code represented by an expression tree.

The Expression Tree does represent the code that would execute for any given construct you create. As long as the environment where you compile and execute the code matches the environment where you create the expression, everything works as expected. When that doesn't happen, the errors are very predictable, and they will be caught in your first tests of any code using the expression trees.

[Next -- Interpreting Expressions](#)

Interpreting Expressions

12/28/2021 • 14 minutes to read • [Edit Online](#)

[Previous -- Executing Expressions](#)

Now, let's write some code to examine the structure of an *expression tree*. Every node in an expression tree will be an object of a class that is derived from `Expression`.

That design makes visiting all the nodes in an expression tree a relatively straight forward recursive operation. The general strategy is to start at the root node and determine what kind of node it is.

If the node type has children, recursively visit the children. At each child node, repeat the process used at the root node: determine the type, and if the type has children, visit each of the children.

Examining an Expression with No Children

Let's start by visiting each node in a simple expression tree. Here's the code that creates a constant expression and then examines its properties:

```
var constant = Expression.Constant(24, typeof(int));

Console.WriteLine($"This is a/an {constant.NodeType} expression type");
Console.WriteLine($"The type of the constant value is {constant.Type}");
Console.WriteLine($"The value of the constant value is {constant.Value}");
```

This will print the following:

```
This is an Constant expression type
The type of the constant value is System.Int32
The value of the constant value is 24
```

Now, let's write the code that would examine this expression and write out some important properties about it. Here's that code:

Examining a simple Addition Expression

Let's start with the addition sample from the introduction to this section.

```
Expression<Func<int>> sum = () => 1 + 2;
```

I'm not using `var` to declare this expression tree, as it is not possible because the right-hand side of the assignment is implicitly typed.

The root node is a `LambdaExpression`. In order to get the interesting code on the right-hand side of the `=>` operator, you need to find one of the children of the `LambdaExpression`. We'll do that with all the expressions in this section. The parent node does help us find the return type of the `LambdaExpression`.

To examine each node in this expression, we'll need to recursively visit a number of nodes. Here's a simple first implementation:

```

Expression<Func<int, int, int>> addition = (a, b) => a + b;

Console.WriteLine($"This expression is a {addition.NodeType} expression type");
Console.WriteLine($"The name of the lambda is {(addition.Name == null) ? "<null>" : addition.Name}");
Console.WriteLine($"The return type is {addition.ReturnType.ToString()}");
Console.WriteLine($"The expression has {addition.Parameters.Count} arguments. They are:");
foreach(var argumentExpression in addition.Parameters)
{
    Console.WriteLine($"Parameter Type: {argumentExpression.Type.ToString()}, Name: {argumentExpression.Name}");
}

var additionBody = (BinaryExpression)addition.Body;
Console.WriteLine($"The body is a {additionBody.NodeType} expression");
Console.WriteLine($"The left side is a {additionBody.Left.NodeType} expression");
var left = (ParameterExpression)additionBody.Left;
Console.WriteLine($"Parameter Type: {left.Type.ToString()}, Name: {left.Name}");
Console.WriteLine($"The right side is a {additionBody.Right.NodeType} expression");
var right = (ParameterExpression)additionBody.Right;
Console.WriteLine($"Parameter Type: {right.Type.ToString()}, Name: {right.Name}");

```

This sample prints the following output:

```

This expression is a/an Lambda expression type
The name of the lambda is <null>
The return type is System.Int32
The expression has 2 arguments. They are:
    Parameter Type: System.Int32, Name: a
    Parameter Type: System.Int32, Name: b
The body is a/an Add expression
The left side is a Parameter expression
    Parameter Type: System.Int32, Name: a
The right side is a Parameter expression
    Parameter Type: System.Int32, Name: b

```

You'll notice a lot of repetition in the code sample above. Let's clean that up and build a more general purpose expression node visitor. That's going to require us to write a recursive algorithm. Any node could be of a type that might have children. Any node that has children requires us to visit those children and determine what that node is. Here's the cleaned up version that utilizes recursion to visit the addition operations:

```

// Base Visitor class:
public abstract class Visitor
{
    private readonly Expression node;

    protected Visitor(Expression node)
    {
        this.node = node;
    }

    public abstract void Visit(string prefix);

    public ExpressionType NodeType => this.node.NodeType;
    public static Visitor CreateFromExpression(Expression node)
    {
        switch(node.NodeType)
        {
            case ExpressionType.Constant:
                return new ConstantVisitor((ConstantExpression)node);
            case ExpressionType.Lambda:
                return new LambdaVisitor((LambdaExpression)node);
            case ExpressionType.Parameter:
                return new ParameterVisitor((ParameterExpression)node);
        }
    }
}

```

```

        case ExpressionType.Add:
            return new BinaryVisitor((BinaryExpression)node);
        default:
            Console.Error.WriteLine($"Node not processed yet: {node.NodeType}");
            return default(Visitor);
    }
}

// Lambda Visitor
public class LambdaVisitor : Visitor
{
    private readonly LambdaExpression node;
    public LambdaVisitor(LambdaExpression node) : base(node)
    {
        this.node = node;
    }

    public override void Visit(string prefix)
    {
        Console.WriteLine($"{prefix}This expression is a {NodeType} expression type");
        Console.WriteLine($"{prefix}The name of the lambda is {(node.Name == null) ? "<null>" : node.Name}");
        Console.WriteLine($"{prefix}The return type is {node.ReturnType.ToString()}");
        Console.WriteLine($"{prefix}The expression has {node.Parameters.Count} argument(s). They are:");
        // Visit each parameter:
        foreach (var argumentExpression in node.Parameters)
        {
            var argumentVisitor = Visitor.CreateFromExpression(argumentExpression);
            argumentVisitor.Visit(prefix + "\t");
        }
        Console.WriteLine($"{prefix}The expression body is:");
        // Visit the body:
        var bodyVisitor = Visitor.CreateFromExpression(node.Body);
        bodyVisitor.Visit(prefix + "\t");
    }
}

// Binary Expression Visitor:
public class BinaryVisitor : Visitor
{
    private readonly BinaryExpression node;
    public BinaryVisitor(BinaryExpression node) : base(node)
    {
        this.node = node;
    }

    public override void Visit(string prefix)
    {
        Console.WriteLine($"{prefix}This binary expression is a {NodeType} expression");
        var left = Visitor.CreateFromExpression(node.Left);
        Console.WriteLine($"{prefix}The Left argument is:");
        left.Visit(prefix + "\t");
        var right = Visitor.CreateFromExpression(node.Right);
        Console.WriteLine($"{prefix}The Right argument is:");
        right.Visit(prefix + "\t");
    }
}

// Parameter visitor:
public class ParameterVisitor : Visitor
{
    private readonly ParameterExpression node;
    public ParameterVisitor(ParameterExpression node) : base(node)
    {
        this.node = node;
    }

    public override void Visit(string prefix)

```

```

    {
        Console.WriteLine($"{prefix}This is an {NodeType} expression type");
        Console.WriteLine($"{prefix}Type: {node.Type.ToString()}, Name: {node.Name}, ByRef:
{node.IsByRef}");
    }
}

// Constant visitor:
public class ConstantVisitor : Visitor
{
    private readonly ConstantExpression node;
    public ConstantVisitor(ConstantExpression node) : base(node)
    {
        this.node = node;
    }

    public override void Visit(string prefix)
    {
        Console.WriteLine($"{prefix}This is an {NodeType} expression type");
        Console.WriteLine($"{prefix}The type of the constant value is {node.Type}");
        Console.WriteLine($"{prefix}The value of the constant value is {node.Value}");
    }
}

```

This algorithm is the basis of an algorithm that can visit any arbitrary `LambdaExpression`. There are many holes, namely that the code I created only looks for a very small sample of the possible sets of expression tree nodes that it may encounter. However, you can still learn quite a bit from what it produces. (The default case in the `Visitor.CreateFromExpression` method prints a message to the error console when a new node type is encountered. That way, you know to add a new expression type.)

When you run this visitor on the addition expression shown above, you get the following output:

```

This expression is a/an Lambda expression type
The name of the lambda is <null>
The return type is System.Int32
The expression has 2 argument(s). They are:
    This is an Parameter expression type
    Type: System.Int32, Name: a, ByRef: False
    This is an Parameter expression type
    Type: System.Int32, Name: b, ByRef: False
The expression body is:
    This binary expression is a Add expression
    The Left argument is:
        This is an Parameter expression type
        Type: System.Int32, Name: a, ByRef: False
    The Right argument is:
        This is an Parameter expression type
        Type: System.Int32, Name: b, ByRef: False

```

Now that you've built a more general visitor implementation, you can visit and process many more different types of expressions.

Examining an Addition Expression with Many Levels

Let's try a more complicated example, yet still limit the node types to addition only:

```
Expression<Func<int>> sum = () => 1 + 2 + 3 + 4;
```

Before you run this on the visitor algorithm, try a thought exercise to work out what the output might be. Remember that the `+` operator is a *binary operator*: it must have two children, representing the left and right operands. There are several possible ways to construct a tree that could be correct:

```

Expression<Func<int>> sum1 = () => 1 + (2 + (3 + 4));
Expression<Func<int>> sum2 = () => ((1 + 2) + 3) + 4;

Expression<Func<int>> sum3 = () => (1 + 2) + (3 + 4);
Expression<Func<int>> sum4 = () => 1 + ((2 + 3) + 4);
Expression<Func<int>> sum5 = () => (1 + (2 + 3)) + 4;

```

You can see the separation into two possible answers to highlight the most promising. The first represents *right associative* expressions. The second represent *left associative* expressions. The advantage of both of those two formats is that the format scales to any arbitrary number of addition expressions.

If you do run this expression through the visitor, you will see this output, verifying that the simple addition expression is *left associative*.

In order to run this sample, and see the full expression tree, I had to make one change to the source expression tree. When the expression tree contains all constants, the resulting tree simply contains the constant value of `10`. The compiler performs all the addition and reduces the expression to its simplest form. Simply adding one variable in the expression is sufficient to see the original tree:

```

Expression<Func<int, int>> sum = (a) => 1 + a + 3 + 4;

```

Create a visitor for this sum and run the visitor you'll see this output:

```

This expression is a/an Lambda expression type
The name of the lambda is <null>
The return type is System.Int32
The expression has 1 argument(s). They are:
    This is an Parameter expression type
    Type: System.Int32, Name: a, ByRef: False
The expression body is:
    This binary expression is a Add expression
    The Left argument is:
        This binary expression is a Add expression
        The Left argument is:
            This binary expression is a Add expression
            The Left argument is:
                This is an Constant expression type
                The type of the constant value is System.Int32
                The value of the constant value is 1
            The Right argument is:
                This is an Parameter expression type
                Type: System.Int32, Name: a, ByRef: False
        The Right argument is:
            This is an Constant expression type
            The type of the constant value is System.Int32
            The value of the constant value is 3
    The Right argument is:
        This is an Constant expression type
        The type of the constant value is System.Int32
        The value of the constant value is 4

```

You can also run any of the other samples through the visitor code and see what tree it represents. Here's an example of the `sum3` expression above (with an additional parameter to prevent the compiler from computing the constant):

```

Expression<Func<int, int, int>> sum3 = (a, b) => (1 + a) + (3 + b);

```

Here's the output from the visitor:

```

This expression is a/an Lambda expression type
The name of the lambda is <null>
The return type is System.Int32
The expression has 2 argument(s). They are:
    This is an Parameter expression type
    Type: System.Int32, Name: a, ByRef: False
    This is an Parameter expression type
    Type: System.Int32, Name: b, ByRef: False
The expression body is:
    This binary expression is a Add expression
    The Left argument is:
        This binary expression is a Add expression
        The Left argument is:
            This is an Constant expression type
            The type of the constant value is System.Int32
            The value of the constant value is 1
        The Right argument is:
            This is an Parameter expression type
            Type: System.Int32, Name: a, ByRef: False
    The Right argument is:
        This binary expression is a Add expression
        The Left argument is:
            This is an Constant expression type
            The type of the constant value is System.Int32
            The value of the constant value is 3
        The Right argument is:
            This is an Parameter expression type
            Type: System.Int32, Name: b, ByRef: False

```

Notice that the parentheses are not part of the output. There are no nodes in the expression tree that represent the parentheses in the input expression. The structure of the expression tree contains all the information necessary to communicate the precedence.

Extending from this sample

The sample deals with only the most rudimentary expression trees. The code you've seen in this section only handles constant integers and the binary `+` operator. As a final sample, let's update the visitor to handle a more complicated expression. Let's make it work for this:

```

Expression<Func<int, int>> factorial = (n) =>
    n == 0 ?
    1 :
    Enumerable.Range(1, n).Aggregate((product, factor) => product * factor);

```

This code represents one possible implementation for the mathematical *factorial* function. The way I've written this code highlights two limitations of building expression trees by assigning lambda expressions to Expressions. First, statement lambdas are not allowed. That means I can't use loops, blocks, if / else statements, and other control structures common in C#. I'm limited to using expressions. Second, I can't recursively call the same expression. I could if it were already a delegate, but I can't call it in its expression tree form. In the section on [building expression trees](#), you'll learn techniques to overcome these limitations.

In this expression, you'll encounter nodes of all these types:

1. Equal (binary expression)
2. Multiply (binary expression)
3. Conditional (the `? :` expression)
4. Method Call Expression (calling `Range()` and `Aggregate()`)

One way to modify the visitor algorithm is to keep executing it, and write the node type every time you reach your `default` clause. After a few iterations, you'll have seen each of the potential nodes. Then, you have all you need. The result would be something like this:

```
public static Visitor CreateFromExpression(Expression node)
{
    switch(node.NodeType)
    {
        case ExpressionType.Constant:
            return new ConstantVisitor((ConstantExpression)node);
        case ExpressionType.Lambda:
            return new LambdaVisitor((LambdaExpression)node);
        case ExpressionType.Parameter:
            return new ParameterVisitor((ParameterExpression)node);
        case ExpressionType.Add:
        case ExpressionType.Equal:
        case ExpressionType.Multiply:
            return new BinaryVisitor((BinaryExpression)node);
        case ExpressionType.Conditional:
            return new ConditionalVisitor((ConditionalExpression)node);
        case ExpressionType.Call:
            return new MethodCallVisitor((MethodCallExpression)node);
        default:
            Console.Error.WriteLine($"Node not processed yet: {node.NodeType}");
            return default(Visitor);
    }
}
```

The `ConditionalVisitor` and `MethodCallVisitor` process those two nodes:

```

public class ConditionalVisitor : Visitor
{
    private readonly ConditionalExpression node;
    public ConditionalVisitor(ConditionalExpression node) : base(node)
    {
        this.node = node;
    }

    public override void Visit(string prefix)
    {
        Console.WriteLine($"{prefix}This expression is a {NodeType} expression");
        var testVisitor = Visitor.CreateFromExpression(node.Test);
        Console.WriteLine($"{prefix}The Test for this expression is:");
        testVisitor.Visit(prefix + "\t");
        var trueVisitor = Visitor.CreateFromExpression(node.IfTrue);
        Console.WriteLine($"{prefix}The True clause for this expression is:");
        trueVisitor.Visit(prefix + "\t");
        var falseVisitor = Visitor.CreateFromExpression(node.IfFalse);
        Console.WriteLine($"{prefix}The False clause for this expression is:");
        falseVisitor.Visit(prefix + "\t");
    }
}

public class MethodCallVisitor : Visitor
{
    private readonly MethodCallExpression node;
    public MethodCallVisitor(MethodCallExpression node) : base(node)
    {
        this.node = node;
    }

    public override void Visit(string prefix)
    {
        Console.WriteLine($"{prefix}This expression is a {NodeType} expression");
        if (node.Object == null)
            Console.WriteLine($"{prefix}This is a static method call");
        else
        {
            Console.WriteLine($"{prefix}The receiver (this) is:");
            var receiverVisitor = Visitor.CreateFromExpression(node.Object);
            receiverVisitor.Visit(prefix + "\t");
        }

        var methodInfo = node.Method;
        Console.WriteLine($"{prefix}The method name is {methodInfo.DeclaringType}.{methodInfo.Name}");
        // There is more here, like generic arguments, and so on.
        Console.WriteLine($"{prefix}The Arguments are:");
        foreach (var arg in node.Arguments)
        {
            var argVisitor = Visitor.CreateFromExpression(arg);
            argVisitor.Visit(prefix + "\t");
        }
    }
}

```

And the output for the expression tree would be:

```

This expression is a/an Lambda expression type
The name of the lambda is <null>
The return type is System.Int32
The expression has 1 argument(s). They are:
    This is an Parameter expression type
    Type: System.Int32, Name: n, ByRef: False
The expression body is:
    This expression is a Conditional expression
    The Test for this expression is:
        This binary expression is a Equal expression
        The Left argument is:
            This is an Parameter expression type
            Type: System.Int32, Name: n, ByRef: False
        The Right argument is:
            This is an Constant expression type
            The type of the constant value is System.Int32
            The value of the constant value is 0
    The True clause for this expression is:
        This is an Constant expression type
        The type of the constant value is System.Int32
        The value of the constant value is 1
    The False clause for this expression is:
        This expression is a Call expression
        This is a static method call
        The method name is System.Linq.Enumerable.Aggregate
        The Arguments are:
            This expression is a Call expression
            This is a static method call
            The method name is System.Linq.Enumerable.Range
            The Arguments are:
                This is an Constant expression type
                The type of the constant value is System.Int32
                The value of the constant value is 1
                This is an Parameter expression type
                Type: System.Int32, Name: n, ByRef: False
            This expression is a Lambda expression type
            The name of the lambda is <null>
            The return type is System.Int32
            The expression has 2 arguments. They are:
                This is an Parameter expression type
                Type: System.Int32, Name: product, ByRef: False
                This is an Parameter expression type
                Type: System.Int32, Name: factor, ByRef: False
            The expression body is:
                This binary expression is a Multiply expression
                The Left argument is:
                    This is an Parameter expression type
                    Type: System.Int32, Name: product, ByRef: False
                The Right argument is:
                    This is an Parameter expression type
                    Type: System.Int32, Name: factor, ByRef: False

```

Extending the Sample Library

The samples in this section show the core techniques to visit and examine nodes in an expression tree. I glossed over many actions you might need in order to concentrate on the core tasks of visiting and accessing nodes in an expression tree.

First, the visitors only handle constants that are integers. Constant values could be any other numeric type, and the C# language supports conversions and promotions between those types. A more robust version of this code would mirror all those capabilities.

Even the last example recognizes a subset of the possible node types. You can still feed it many expressions that will cause it to fail. A full implementation is included in .NET Standard under the name [ExpressionVisitor](#) and can

handle all the possible node types.

Finally, the library I used in this article was built for demonstration and learning. It's not optimized. I wrote it to make the structures used clear, and to highlight the techniques used to visit the nodes and analyze what's there. A production implementation would pay more attention to performance than I have.

Even with those limitations, you should be well on your way to writing algorithms that read and understand expression trees.

[Next -- Building Expressions](#)

Building Expression Trees

12/28/2021 • 5 minutes to read • [Edit Online](#)

[Previous -- Interpreting Expressions](#)

All the expression trees you've seen so far have been created by the C# compiler. All you had to do was create a lambda expression that was assigned to a variable typed as an `Expression<Func<T>>` or some similar type. That's not the only way to create an expression tree. For many scenarios you may find that you need to build an expression in memory at run time.

Building Expression Trees is complicated by the fact that those expression trees are immutable. Being immutable means that you must build the tree from the leaves up to the root. The APIs you'll use to build expression trees reflect this fact: The methods you'll use to build a node take all its children as arguments. Let's walk through a few examples to show you the techniques.

Creating Nodes

Let's start relatively simply again. We'll use the addition expression I've been working with throughout these sections:

```
Expression<Func<int>> sum = () => 1 + 2;
```

To construct that expression tree, you must construct the leaf nodes. The leaf nodes are constants, so you can use the `Expression.Constant` method to create the nodes:

```
var one = Expression.Constant(1, typeof(int));  
var two = Expression.Constant(2, typeof(int));
```

Next, you'll build the addition expression:

```
var addition = Expression.Add(one, two);
```

Once you've got the addition expression, you can create the lambda expression:

```
var lambda = Expression.Lambda(addition);
```

This is a very simple lambda expression, because it contains no arguments. Later in this section, you'll see how to map arguments to parameters and build more complicated expressions.

For expressions that are as simple as this one, you may combine all the calls into a single statement:

```
var lambda = Expression.Lambda(  
    Expression.Add(  
        Expression.Constant(1, typeof(int)),  
        Expression.Constant(2, typeof(int))  
    )  
);
```

Building a Tree

That's the basics of building an expression tree in memory. More complex trees generally mean more node types, and more nodes in the tree. Let's run through one more example and show two more node types that you will typically build when you create expression trees: the argument nodes, and method call nodes.

Let's build an expression tree to create this expression:

```
Expression<Func<double, double, double>> distanceCalc =  
    (x, y) => Math.Sqrt(x * x + y * y);
```

You'll start by creating parameter expressions for `x` and `y`:

```
var xParameter = Expression.Parameter(typeof(double), "x");  
var yParameter = Expression.Parameter(typeof(double), "y");
```

Creating the multiplication and addition expressions follows the pattern you've already seen:

```
var xSquared = Expression.Multiply(xParameter, xParameter);  
var ySquared = Expression.Multiply(yParameter, yParameter);  
var sum = Expression.Add(xSquared, ySquared);
```

Next, you need to create a method call expression for the call to `Math.Sqrt`.

```
var sqrtMethod = typeof(Math).GetMethod("Sqrt", new[] { typeof(double) });  
var distance = Expression.Call(sqrtMethod, sum);
```

And then finally, you put the method call into a lambda expression, and make sure to define the arguments to the lambda expression:

```
var distanceLambda = Expression.Lambda(  
    distance,  
    xParameter,  
    yParameter);
```

In this more complicated example, you see a couple more techniques that you will often need to create expression trees.

First, you need to create the objects that represent parameters or local variables before you use them. Once you've created those objects, you can use them in your expression tree wherever you need.

Second, you need to use a subset of the Reflection APIs to create a `MethodInfo` object so that you can create an expression tree to access that method. You must limit yourself to the subset of the Reflection APIs that are available on the .NET Core platform. Again, these techniques will extend to other expression trees.

Building Code In Depth

You aren't limited in what you can build using these APIs. However, the more complicated expression tree that you want to build, the more difficult the code is to manage and to read.

Let's build an expression tree that is the equivalent of this code:

```
Func<int, int> factorialFunc = (n) =>
{
    var res = 1;
    while (n > 1)
    {
        res = res * n;
        n--;
    }
    return res;
};
```

Notice above that I did not build the expression tree, but simply the delegate. Using the `Expression` class, you can't build statement lambdas. Here's the code that is required to build the same functionality. It's complicated by the fact that there isn't an API to build a `while` loop, instead you need to build a loop that contains a conditional test, and a label target to break out of the loop.

```
var nArgument = Expression.Parameter(typeof(int), "n");
var result = Expression.Variable(typeof(int), "result");

// Creating a label that represents the return value
LabelTarget label = Expression.Label(typeof(int));

var initializeResult = Expression.Assign(result, Expression.Constant(1));

// This is the inner block that performs the multiplication,
// and decrements the value of 'n'
var block = Expression.Block(
    Expression.Assign(result,
        Expression.Multiply(result, nArgument)),
    Expression.PostDecrementAssign(nArgument)
);

// Creating a method body.
BlockExpression body = Expression.Block(
    new[] { result },
    initializeResult,
    Expression.Loop(
        Expression.IfThenElse(
            Expression.GreaterThan(nArgument, Expression.Constant(1)),
            block,
            Expression.Break(label, result)
        ),
        label
    )
);
```

The code to build the expression tree for the factorial function is quite a bit longer, more complicated, and it's riddled with labels and break statements and other elements we'd like to avoid in our everyday coding tasks.

For this section, I've also updated the visitor code to visit every node in this expression tree and write out information about the nodes that are created in this sample. You can [view or download the sample code](#) at the dotnet/docs GitHub repository. Experiment for yourself by building and running the samples. For download instructions, see [Samples and Tutorials](#).

Examining the APIs

The expression tree APIs are some of the more difficult to navigate in .NET Core, but that's fine. Their purpose is a rather complex undertaking: writing code that generates code at run time. They are necessarily complicated to provide a balance between supporting all the control structures available in the C# language and keeping the surface area of the APIs as small as reasonable. This balance means that many control structures are

represented not by their C# constructs, but by constructs that represent the underlying logic that the compiler generates from these higher level constructs.

Also, at this time, there are C# expressions that cannot be built directly using `Expression` class methods. In general, these will be the newest operators and expressions added in C# 5 and C# 6. (For example, `async` expressions cannot be built, and the new `?.` operator cannot be directly created.)

[Next -- Translating Expressions](#)

Translate expression trees

12/28/2021 • 6 minutes to read • [Edit Online](#)

[Previous -- Building Expressions](#)

In this final section, you'll learn how to visit each node in an expression tree while building a modified copy of that expression tree. These are the techniques that you will use in two important scenarios. The first is to understand the algorithms expressed by an expression tree so that it can be translated into another environment. The second is when you want to change the algorithm that has been created. This might be to add logging, intercept method calls and track them, or other purposes.

Translating is Visiting

The code you build to translate an expression tree is an extension of what you've already seen to visit all the nodes in a tree. When you translate an expression tree, you visit all the nodes, and while visiting them, build the new tree. The new tree may contain references to the original nodes, or new nodes that you have placed in the tree.

Let's see this in action by visiting an expression tree, and creating a new tree with some replacement nodes. In this example, let's replace any constant with a constant that is ten times larger. Otherwise, we'll leave the expression tree intact. Rather than reading the value of the constant, and replacing it with a new constant, we'll make this replacement by replacing the constant node with a new node that performs the multiplication.

Here, once you find a constant node, you create a new multiplication node whose children are the original constant, and the constant `10`:

```
private static Expression ReplaceNodes(Expression original)
{
    if (original.NodeType == ExpressionType.Constant)
    {
        return Expression.Multiply(original, Expression.Constant(10));
    }
    else if (original.NodeType == ExpressionType.Add)
    {
        var binaryExpression = (BinaryExpression)original;
        return Expression.Add(
            ReplaceNodes(binaryExpression.Left),
            ReplaceNodes(binaryExpression.Right));
    }
    return original;
}
```

By replacing the original node with the substitute, a new tree is formed that contains our modifications. We can verify that by compiling and executing the replaced tree.

```
var one = Expression.Constant(1, typeof(int));
var two = Expression.Constant(2, typeof(int));
var addition = Expression.Add(one, two);
var sum = ReplaceNodes(addition);
var executableFunc = Expression.Lambda(sum);

var func = (Func<int>)executableFunc.Compile();
var answer = func();
Console.WriteLine(answer);
```

Building a new tree is a combination of visiting the nodes in the existing tree, and creating new nodes and inserting them into the tree.

This example shows the importance of expression trees being immutable. Notice that the new tree created above contains a mixture of newly created nodes, and nodes from the existing tree. That's safe, because the nodes in the existing tree cannot be modified. This can result in significant memory efficiencies. The same nodes can be used throughout a tree, or in multiple expression trees. Since nodes can't be modified, the same node can be reused whenever it's needed.

Traversing and Executing an Addition

Let's verify this by building a second visitor that walks the tree of addition nodes and computes the result. You can do this by making a couple modifications to the visitor that you've seen so far. In this new version, the visitor will return the partial sum of the addition operation up to this point. For a constant expression, that is simply the value of the constant expression. For an addition expression, the result is the sum of the left and right operands, once those trees have been traversed.

```
var one = Expression.Constant(1, typeof(int));
var two = Expression.Constant(2, typeof(int));
var three = Expression.Constant(3, typeof(int));
var four = Expression.Constant(4, typeof(int));
var addition = Expression.Add(one, two);
var add2 = Expression.Add(three, four);
var sum = Expression.Add(addition, add2);

// Declare the delegate, so we can call it
// from itself recursively:
Func<Expression, int> aggregate = null;
// Aggregate, return constants, or the sum of the left and right operand.
// Major simplification: Assume every binary expression is an addition.
aggregate = (exp) =>
    exp.NodeType == ExpressionType.Constant ?
        (int)((ConstantExpression)exp).Value :
        aggregate(((BinaryExpression)exp).Left) + aggregate(((BinaryExpression)exp).Right);

var theSum = aggregate(sum);
Console.WriteLine(theSum);
```

There's quite a bit of code here, but the concepts are very approachable. This code visits children in a depth first search. When it encounters a constant node, the visitor returns the value of the constant. After the visitor has visited both children, those children will have computed the sum computed for that subtree. The addition node can now compute its sum. Once all the nodes in the expression tree have been visited, the sum will have been computed. You can trace the execution by running the sample in the debugger and tracing the execution.

Let's make it easier to trace how the nodes are analyzed and how the sum is computed by traversing the tree. Here's an updated version of the Aggregate method that includes quite a bit of tracing information:

```

private static int Aggregate(Expression exp)
{
    if (exp.NodeType == ExpressionType.Constant)
    {
        var constantExp = (ConstantExpression)exp;
        Console.Error.WriteLine($"Found Constant: {constantExp.Value}");
        return (int)constantExp.Value;
    }
    else if (exp.NodeType == ExpressionType.Add)
    {
        var addExp = (BinaryExpression)exp;
        Console.Error.WriteLine("Found Addition Expression");
        Console.Error.WriteLine("Computing Left node");
        var leftOperand = Aggregate(addExp.Left);
        Console.Error.WriteLine($"Left is: {leftOperand}");
        Console.Error.WriteLine("Computing Right node");
        var rightOperand = Aggregate(addExp.Right);
        Console.Error.WriteLine($"Right is: {rightOperand}");
        var sum = leftOperand + rightOperand;
        Console.Error.WriteLine($"Computed sum: {sum}");
        return sum;
    }
    else throw new NotSupportedException("Haven't written this yet");
}

```

Running it on the same expression yields the following output:

```

10
Found Addition Expression
Computing Left node
Found Addition Expression
Computing Left node
Found Constant: 1
Left is: 1
Computing Right node
Found Constant: 2
Right is: 2
Computed sum: 3
Left is: 3
Computing Right node
Found Addition Expression
Computing Left node
Found Constant: 3
Left is: 3
Computing Right node
Found Constant: 4
Right is: 4
Computed sum: 7
Right is: 7
Computed sum: 10
10

```

Trace the output and follow along in the code above. You should be able to work out how the code visits each node and computes the sum as it goes through the tree and finds the sum.

Now, let's look at a different run, with the expression given by `sum1`:

```
Expression<Func<int> sum1 = () => 1 + (2 + (3 + 4));
```

Here's the output from examining this expression:

```
Found Addition Expression
Computing Left node
Found Constant: 1
Left is: 1
Computing Right node
Found Addition Expression
Computing Left node
Found Constant: 2
Left is: 2
Computing Right node
Found Addition Expression
Computing Left node
Found Constant: 3
Left is: 3
Computing Right node
Found Constant: 4
Right is: 4
Computed sum: 7
Right is: 7
Computed sum: 9
Right is: 9
Computed sum: 10
10
```

While the final answer is the same, the tree traversal is completely different. The nodes are traveled in a different order, because the tree was constructed with different operations occurring first.

Learning More

This sample shows a small subset of the code you would build to traverse and interpret the algorithms represented by an expression tree. For a complete discussion of all the work necessary to build a general purpose library that translates expression trees into another language, please read [this series](#) by Matt Warren. It goes into great detail on how to translate any of the code you might find in an expression tree.

I hope you've now seen the true power of expression trees. You can examine a set of code, make any changes you'd like to that code, and execute the changed version. Because the expression trees are immutable, you can create new trees by using the components of existing trees. This minimizes the amount of memory needed to create modified expression trees.

[Next -- Summing up](#)

Expression Trees Summary

12/28/2021 • 2 minutes to read • [Edit Online](#)

[Previous -- Translating Expressions](#)

In this series, you've seen how you can use *expression trees* to create dynamic programs that interpret code as data and build new functionality based on that code.

You can examine expression trees to understand the intent of an algorithm. You can not only examine that code. You can build new expression trees that represent modified versions of the original code.

You can also use expression trees to look at an algorithm, and translate that algorithm into another language or environment.

Limitations

There are some newer C# language elements that don't translate well into expression trees. Expression trees cannot contain `await` expressions, or `async` lambda expressions. Many of the features added in the C# 6 release don't appear exactly as written in expression trees. Instead, newer features will be exposed in expressions trees in the equivalent, earlier syntax. This may not be as much of a limitation as you might think. In fact, it means that your code that interprets expression trees will likely still work the same when new language features are introduced.

Even with these limitations, expression trees do enable you to create dynamic algorithms that rely on interpreting and modifying code that is represented as a data structure. It's a powerful tool, and it's one of the features of the .NET ecosystem that enables rich libraries such as Entity Framework to accomplish what they do.

Interoperability (C# Programming Guide)

12/28/2021 • 2 minutes to read • [Edit Online](#)

Interoperability enables you to preserve and take advantage of existing investments in unmanaged code. Code that runs under the control of the common language runtime (CLR) is called *managed code*, and code that runs outside the CLR is called *unmanaged code*. COM, COM+, C++ components, ActiveX components, and Microsoft Windows API are examples of unmanaged code.

.NET enables interoperability with unmanaged code through platform invoke services, the [System.Runtime.InteropServices](#) namespace, C++ interoperability, and COM interoperability (COM interop).

In This Section

[Interoperability Overview](#)

Describes methods to interoperate between C# managed code and unmanaged code.

[How to access Office interop objects by using C# features](#)

Describes features that are introduced in Visual C# to facilitate Office programming.

[How to use indexed properties in COM interop programming](#)

Describes how to use indexed properties to access COM properties that have parameters.

[How to use platform invoke to play a WAV file](#)

Describes how to use platform invoke services to play a .wav sound file on the Windows operating system.

[Walkthrough: Office Programming](#)

Shows how to create an Excel workbook and a Word document that contains a link to the workbook.

[Example COM Class](#)

Demonstrates how to expose a C# class as a COM object.

C# Language Specification

For more information, see [Basic concepts](#) in the [C# Language Specification](#). The language specification is the definitive source for C# syntax and usage.

See also

- [Marshal.ReleaseComObject](#)
- [C# Programming Guide](#)
- [Interoperating with Unmanaged Code](#)
- [Walkthrough: Office Programming](#)

Versioning in C#

12/28/2021 • 5 minutes to read • [Edit Online](#)

In this tutorial you'll learn what versioning means in .NET. You'll also learn the factors to consider when versioning your library as well as upgrading to a new version of a library.

Authoring Libraries

As a developer who has created .NET libraries for public use, you've most likely been in situations where you have to roll out new updates. How you go about this process matters a lot as you need to ensure that there's a seamless transition of existing code to the new version of your library. Here are several things to consider when creating a new release:

Semantic Versioning

Semantic versioning (SemVer for short) is a naming convention applied to versions of your library to signify specific milestone events. Ideally, the version information you give your library should help developers determine the compatibility with their projects that make use of older versions of that same library.

The most basic approach to SemVer is the 3 component format `MAJOR.MINOR.PATCH`, where:

- `MAJOR` is incremented when you make incompatible API changes
- `MINOR` is incremented when you add functionality in a backwards-compatible manner
- `PATCH` is incremented when you make backwards-compatible bug fixes

There are also ways to specify other scenarios like pre-release versions etc. when applying version information to your .NET library.

Backwards Compatibility

As you release new versions of your library, backwards compatibility with previous versions will most likely be one of your major concerns. A new version of your library is source compatible with a previous version if code that depends on the previous version can, when recompiled, work with the new version. A new version of your library is binary compatible if an application that depended on the old version can, without recompilation, work with the new version.

Here are some things to consider when trying to maintain backwards compatibility with older versions of your library:

- **Virtual methods:** When you make a virtual method non-virtual in your new version it means that projects that override that method will have to be updated. This is a huge breaking change and is strongly discouraged.
- **Method signatures:** When updating a method behavior requires you to change its signature as well, you should instead create an overload so that code calling into that method will still work. You can always manipulate the old method signature to call into the new method signature so that implementation remains consistent.
- **Obsolete attribute:** You can use this attribute in your code to specify classes or class members that are deprecated and likely to be removed in future versions. This ensures developers utilizing your library are better prepared for breaking changes.
- **Optional Method Arguments:** When you make previously optional method arguments compulsory or change their default value then all code that does not supply those arguments will need to be updated.

NOTE

Making compulsory arguments optional should have very little effect especially if it doesn't change the method's behavior.

The easier you make it for your users to upgrade to the new version of your library, the more likely that they will upgrade sooner.

Application Configuration File

As a .NET developer there's a very high chance you've encountered the `app.config` file present in most project types. This simple configuration file can go a long way into improving the rollout of new updates. You should generally design your libraries in such a way that information that is likely to change regularly is stored in the `app.config` file, this way when such information is updated, the config file of older versions just needs to be replaced with the new one without the need for recompilation of the library.

Consuming Libraries

As a developer that consumes .NET libraries built by other developers you're most likely aware that a new version of a library might not be fully compatible with your project and you might often find yourself having to update your code to work with those changes.

Lucky for you, C# and the .NET ecosystem comes with features and techniques that allow us to easily update our app to work with new versions of libraries that might introduce breaking changes.

Assembly Binding Redirection

You can use the `app.config` file to update the version of a library your app uses. By adding what is called a [binding redirect](#), you can use the new library version without having to recompile your app. The following example shows how you would update your app's `app.config` file to use the `1.0.1` patch version of `ReferencedLibrary` instead of the `1.0.0` version it was originally compiled with.

```
<dependentAssembly>
  <assemblyIdentity name="ReferencedLibrary" publicKeyToken="32ab4ba45e0a69a1" culture="en-us" />
  <bindingRedirect oldVersion="1.0.0" newVersion="1.0.1" />
</dependentAssembly>
```

NOTE

This approach will only work if the new version of `ReferencedLibrary` is binary compatible with your app. See the [Backwards Compatibility](#) section above for changes to look out for when determining compatibility.

new

You use the `new` modifier to hide inherited members of a base class. This is one way derived classes can respond to updates in base classes.

Take the following example:


```

public class BaseClass
{
    public void MyMethod()
    {
        Console.WriteLine("A base method");
    }
}

public class DerivedClass : BaseClass
{
    public new void MyMethod()
    {
        Console.WriteLine("A derived method");
    }
}

public static void Main()
{
    BaseClass b = new BaseClass();
    DerivedClass d = new DerivedClass();

    b.MyMethod();
    d.MyMethod();
}

```

Output

```

A base method
A derived method

```

From the example above you can see how `DerivedClass` hides the `MyMethod` method present in `BaseClass`. This means that when a base class in the new version of a library adds a member that already exists in your derived class, you can simply use the `new` modifier on your derived class member to hide the base class member.

When no `new` modifier is specified, a derived class will by default hide conflicting members in a base class, although a compiler warning will be generated the code will still compile. This means that simply adding new members to an existing class makes that new version of your library both source and binary compatible with code that depends on it.

override

The `override` modifier means a derived implementation extends the implementation of a base class member rather than hides it. The base class member needs to have the `virtual` modifier applied to it.

```
public class MyBaseClass
{
    public virtual string MethodOne()
    {
        return "Method One";
    }
}

public class MyDerivedClass : MyBaseClass
{
    public override string MethodOne()
    {
        return "Derived Method One";
    }
}

public static void Main()
{
    MyBaseClass b = new MyBaseClass();
    MyDerivedClass d = new MyDerivedClass();

    Console.WriteLine("Base Method One: {0}", b.MethodOne());
    Console.WriteLine("Derived Method One: {0}", d.MethodOne());
}
```

Output

```
Base Method One: Method One
Derived Method One: Derived Method One
```

The `override` modifier is evaluated at compile time and the compiler will throw an error if it doesn't find a virtual member to override.

Your knowledge of the discussed techniques and your understanding of the situations in which to use them, will go a long way towards easing the transition between versions of a library.

How to (C#)

12/28/2021 • 3 minutes to read • [Edit Online](#)

In the How to section of the C# Guide, you can find quick answers to common questions. In some cases, articles may be listed in multiple sections. We wanted to make them easy to find for multiple search paths.

General C# concepts

There are several tips and tricks that are common C# developer practices:

- [Initialize objects using an object initializer.](#)
- [Learn the differences between passing a struct and a class to a method.](#)
- [Use operator overloading.](#)
- [Implement and call a custom extension method.](#)
- [Create a new method for an `enum` type using extension methods.](#)

Class, record, and struct members

You create classes, records, and structs to implement your program. These techniques are commonly used when writing classes, records, or structs.

- [Declare auto implemented properties.](#)
- [Declare and use read/write properties.](#)
- [Define constants.](#)
- [Override the `ToString` method to provide string output.](#)
- [Define abstract properties.](#)
- [Use the xml documentation features to document your code.](#)
- [Explicitly implement interface members](#) to keep your public interface concise.
- [Explicitly implement members of two interfaces.](#)

Working with collections

These articles help you work with collections of data.

- [Initialize a dictionary with a collection initializer.](#)

Working with strings

Strings are the fundamental data type used to display or manipulate text. These articles demonstrate common practices with strings.

- [Compare strings.](#)
- [Modify the contents of a string.](#)
- [Determine if a string represents a number.](#)
- [Use `String.Split` to separate strings.](#)
- [Combine multiple strings into one.](#)
- [Search for text in a string.](#)

Convert between types

You may need to convert an object to a different type.

- [Determine if a string represents a number.](#)
- [Convert between strings that represent hexadecimal numbers and the number.](#)
- [Convert a string to a `DateTime`.](#)
- [Convert a byte array to an int.](#)
- [Convert a string to a number.](#)
- [Use pattern matching, the `as` and `is` operators to safely cast to a different type.](#)
- [Define custom type conversions.](#)
- [Determine if a type is a nullable value type.](#)
- [Convert between nullable and non-nullable value types.](#)

Equality and ordering comparisons

You may create types that define their own rules for equality or define a natural ordering among objects of that type.

- [Test for reference-based equality.](#)
- [Define value-based equality for a type.](#)

Exception handling

.NET programs report that methods did not successfully complete their work by throwing exceptions. In these articles you'll learn to work with exceptions.

- [Handle exceptions using `try` and `catch`.](#)
- [Cleanup resources using `finally` clauses.](#)
- [Recover from non-CLS \(Common Language Specification\) exceptions.](#)

Delegates and events

Delegates and events provide a capability for strategies that involve loosely coupled blocks of code.

- [Declare, instantiate, and use delegates.](#)
- [Combine multicast delegates.](#)

Events provide a mechanism to publish or subscribe to notifications.

- [Subscribe and unsubscribe from events.](#)
- [Implement events declared in interfaces.](#)
- [Conform to .NET guidelines when your code publishes events.](#)
- [Raise events defined in base classes from derived classes.](#)
- [Implement custom event accessors.](#)

LINQ practices

LINQ enables you to write code to query any data source that supports the LINQ query expression pattern. These articles help you understand the pattern and work with different data sources.

- [Query a collection.](#)
- [Use `var` in query expressions.](#)
- [Return subsets of element properties from a query.](#)
- [Write queries with complex filtering.](#)
- [Sort elements of a data source.](#)
- [Sort elements on multiple keys.](#)

- [Control the type of a projection.](#)
- [Count occurrences of a value in a source sequence.](#)
- [Calculate intermediate values.](#)
- [Merge data from multiple sources.](#)
- [Find the set difference between two sequences.](#)
- [Debug empty query results.](#)
- [Add custom methods to LINQ queries.](#)

Multiple threads and async processing

Modern programs often use asynchronous operations. These articles will help you learn to use these techniques.

- [Improve async performance using `System.Threading.Tasks.Task.WhenAll`.](#)
- [Make multiple web requests in parallel using `async` and `await`.](#)
- [Use a thread pool.](#)

Command line args to your program

Typically, C# programs have command line arguments. These articles teach you to access and process those command line arguments.

- [Retrieve all command line arguments with `for`.](#)

How to separate strings using String.Split in C#

12/28/2021 • 2 minutes to read • [Edit Online](#)

The [String.Split](#) method creates an array of substrings by splitting the input string based on one or more delimiters. This method is often the easiest way to separate a string on word boundaries. It's also used to split strings on other specific characters or strings.

NOTE

The C# examples in this article run in the [Try.NET](#) inline code runner and playground. Select the **Run** button to run an example in an interactive window. Once you execute the code, you can modify it and run the modified code by selecting **Run** again. The modified code either runs in the interactive window or, if compilation fails, the interactive window displays all C# compiler error messages.

The following code splits a common phrase into an array of strings for each word.

```
string phrase = "The quick brown fox jumps over the lazy dog.";
string[] words = phrase.Split(' ');

foreach (var word in words)
{
    System.Console.WriteLine($"<{word}>");
}
```

Every instance of a separator character produces a value in the returned array. Consecutive separator characters produce the empty string as a value in the returned array. You can see how an empty string is created in the following example, which uses the space character as a separator.

```
string phrase = "The quick brown   fox      jumps over the lazy dog.";
string[] words = phrase.Split(' ');

foreach (var word in words)
{
    System.Console.WriteLine($"<{word}>");
}
```

This behavior makes it easier for formats like comma-separated values (CSV) files representing tabular data. Consecutive commas represent a blank column.

You can pass an optional [StringSplitOptions.RemoveEmptyEntries](#) parameter to exclude any empty strings in the returned array. For more complicated processing of the returned collection, you can use [LINQ](#) to manipulate the result sequence.

[String.Split](#) can use multiple separator characters. The following example uses spaces, commas, periods, colons, and tabs as separating characters, which are passed to [Split](#) in an array. The loop at the bottom of the code displays each of the words in the returned array.

```
char[] delimiterChars = { ' ', ',', '.', ':', '\t' };

string text = "one\ttwo three:four,five six seven";
System.Console.WriteLine($"Original text: '{text}'");

string[] words = text.Split(delimiterChars);
System.Console.WriteLine($"{words.Length} words in text:");

foreach (var word in words)
{
    System.Console.WriteLine($"<{word}>");
}
```

Consecutive instances of any separator produce the empty string in the output array:

```
char[] delimiterChars = { ' ', ',', '.', ':', '\t' };

string text = "one\ttwo :,five six seven";
System.Console.WriteLine($"Original text: '{text}'");

string[] words = text.Split(delimiterChars);
System.Console.WriteLine($"{words.Length} words in text:");

foreach (var word in words)
{
    System.Console.WriteLine($"<{word}>");
}
```

[String.Split](#) can take an array of strings (character sequences that act as separators for parsing the target string, instead of single characters).

```
string[] separatingStrings = { "<<", "... " };

string text = "one<<two.....three<four";
System.Console.WriteLine($"Original text: '{text}'");

string[] words = text.Split(separatingStrings, System.StringSplitOptions.RemoveEmptyEntries);
System.Console.WriteLine($"{words.Length} substrings in text:");

foreach (var word in words)
{
    System.Console.WriteLine(word);
}
```

See also

- [Extract elements from a string](#)
- [C# programming guide](#)
- [Strings](#)
- [.NET regular expressions](#)

How to concatenate multiple strings (C# Guide)

12/28/2021 • 4 minutes to read • [Edit Online](#)

Concatenation is the process of appending one string to the end of another string. You concatenate strings by using the `+` operator. For string literals and string constants, concatenation occurs at compile time; no run-time concatenation occurs. For string variables, concatenation occurs only at run time.

NOTE

The C# examples in this article run in the [Try.NET](#) inline code runner and playground. Select the **Run** button to run an example in an interactive window. Once you execute the code, you can modify it and run the modified code by selecting **Run** again. The modified code either runs in the interactive window or, if compilation fails, the interactive window displays all C# compiler error messages.

String literals

The following example splits a long string literal into smaller strings to improve readability in the source code. The code concatenates the smaller strings to create the long string literal. The parts are concatenated into a single string at compile time. There's no run-time performance cost regardless of the number of strings involved.

```
// Concatenation of literals is performed at compile time, not run time.
string text = "Historically, the world of data and the world of objects " +
    "have not been well integrated. Programmers work in C# or Visual Basic " +
    "and also in SQL or XQuery. On the one side are concepts such as classes, " +
    "objects, fields, inheritance, and .NET Framework APIs. On the other side " +
    "are tables, columns, rows, nodes, and separate languages for dealing with " +
    "them. Data types often require translation between the two worlds; there are " +
    "different standard functions. Because the object world has no notion of query, a " +
    "query can only be represented as a string without compile-time type checking or " +
    "IntelliSense support in the IDE. Transferring data from SQL tables or XML trees to " +
    "objects in memory is often tedious and error-prone.";

System.Console.WriteLine(text);
```

`+` and `+=` operators

To concatenate string variables, you can use the `+` or `+=` operators, [string interpolation](#) or the [String.Format](#), [String.Concat](#), [String.Join](#) or [StringBuilder.Append](#) methods. The `+` operator is easy to use and makes for intuitive code. Even if you use several `+` operators in one statement, the string content is copied only once. The following code shows examples of using the `+` and `+=` operators to concatenate strings:

```
string userName = "<Type your name here>";
string dateString = DateTime.Today.ToShortDateString();

// Use the + and += operators for one-time concatenations.
string str = "Hello " + userName + ". Today is " + dateString + ".";
System.Console.WriteLine(str);

str += " How are you today?";
System.Console.WriteLine(str);
```


String interpolation

In some expressions, it's easier to concatenate strings using string interpolation, as the following code shows:

```
string userName = "<Type your name here>";
string date = DateTime.Today.ToShortDateString();

// Use string interpolation to concatenate strings.
string str = $"Hello {userName}. Today is {date}.";
System.Console.WriteLine(str);

str = $" {str} How are you today?";
System.Console.WriteLine(str);
```

NOTE

In string concatenation operations, the C# compiler treats a null string the same as an empty string.

Beginning with C# 10, you can use string interpolation to initialize a constant string when all the expressions used for placeholders are also constant strings.

String.Format

Another method to concatenate strings is [String.Format](#). This method works well when you're building a string from a small number of component strings.

StringBuilder

In other cases, you may be combining strings in a loop where you don't know how many source strings you're combining, and the actual number of source strings may be large. The [StringBuilder](#) class was designed for these scenarios. The following code uses the [Append](#) method of the [StringBuilder](#) class to concatenate strings.

```
// Use StringBuilder for concatenation in tight loops.
var sb = new System.Text.StringBuilder();
for (int i = 0; i < 20; i++)
{
    sb.AppendLine(i.ToString());
}
System.Console.WriteLine(sb.ToString());
```

You can read more about the [reasons to choose string concatenation or the `StringBuilder` class](#).

String.Concat OR String.Join

Another option to join strings from a collection is to use [String.Concat](#) method. Use [String.Join](#) method if source strings should be separated by a delimiter. The following code combines an array of words using both methods:

```
string[] words = { "The", "quick", "brown", "fox", "jumps", "over", "the", "lazy", "dog." };

var unreadablePhrase = string.Concat(words);
System.Console.WriteLine(unreadablePhrase);

var readablePhrase = string.Join(" ", words);
System.Console.WriteLine(readablePhrase);
```

LINQ and `Enumerable.Aggregate`

At last, you can use [LINQ](#) and the [Enumerable.Aggregate](#) method to join strings from a collection. This method combines the source strings using a lambda expression. The lambda expression does the work to add each string to the existing accumulation. The following example combines an array of words, adding a space between each word in the array:

```
string[] words = { "The", "quick", "brown", "fox", "jumps", "over", "the", "lazy", "dog." };

var phrase = words.Aggregate((partialPhrase, word) => $"{partialPhrase} {word}");
System.Console.WriteLine(phrase);
```

This option can cause more allocations than other methods for concatenating collections, as it creates an intermediate string for each iteration. If optimizing performance is critical, consider the [StringBuilder](#) class or the [String.Concat](#) or [String.Join](#) method to concatenate a collection, instead of `Enumerable.Aggregate`.

See also

- [String](#)
- [StringBuilder](#)
- [C# programming guide](#)
- [Strings](#)

How to search strings

12/28/2021 • 4 minutes to read • [Edit Online](#)

You can use two main strategies to search for text in strings. Methods of the [String](#) class search for specific text. Regular expressions search for patterns in text.

NOTE

The C# examples in this article run in the [Try.NET](#) inline code runner and playground. Select the **Run** button to run an example in an interactive window. Once you execute the code, you can modify it and run the modified code by selecting **Run** again. The modified code either runs in the interactive window or, if compilation fails, the interactive window displays all C# compiler error messages.

The [string](#) type, which is an alias for the [System.String](#) class, provides a number of useful methods for searching the contents of a string. Among them are [Contains](#), [StartsWith](#), [EndsWith](#), [IndexOf](#), [LastIndexOf](#). The [System.Text.RegularExpressions.Regex](#) class provides a rich vocabulary to search for patterns in text. In this article, you learn these techniques and how to choose the best method for your needs.

Does a string contain text?

The [String.Contains](#), [String.StartsWith](#), and [String.EndsWith](#) methods search a string for specific text. The following example shows each of these methods and a variation that uses a case-insensitive search:

```
string factMessage = "Extension methods have all the capabilities of regular static methods.";

// Write the string and include the quotation marks.
Console.WriteLine($"\"{factMessage}\"");

// Simple comparisons are always case sensitive!
bool containsSearchResult = factMessage.Contains("extension");
Console.WriteLine($"Contains \"extension\"? {containsSearchResult}");

// For user input and strings that will be displayed to the end user,
// use the StringComparison parameter on methods that have it to specify how to match strings.
bool ignoreCaseSearchResult = factMessage.StartsWith("extension",
    System.StringComparison.CurrentCultureIgnoreCase);
Console.WriteLine($"Starts with \"extension\"? {ignoreCaseSearchResult} (ignoring case)");

bool endsWithSearchResult = factMessage.EndsWith(".", System.StringComparison.CurrentCultureIgnoreCase);
Console.WriteLine($"Ends with '.'? {endsWithSearchResult}");
```

The preceding example demonstrates an important point for using these methods. Searches are **case-sensitive** by default. You use the [StringComparison.CurrentCultureIgnoreCase](#) enumeration value to specify a case-insensitive search.

Where does the sought text occur in a string?

The [IndexOf](#) and [LastIndexOf](#) methods also search for text in strings. These methods return the location of the text being sought. If the text isn't found, they return `-1`. The following example shows a search for the first and last occurrence of the word "methods" and displays the text in between.

```

string factMessage = "Extension methods have all the capabilities of regular static methods.";

// Write the string and include the quotation marks.
Console.WriteLine($"\"{factMessage}\"");

// This search returns the substring between two strings, so
// the first index is moved to the character just after the first string.
int first = factMessage.IndexOf("methods") + "methods".Length;
int last = factMessage.LastIndexOf("methods");
string str2 = factMessage.Substring(first, last - first);
Console.WriteLine($"Substring between \"methods\" and \"methods\": '{str2}'");

```

Finding specific text using regular expressions

The [System.Text.RegularExpressions.Regex](#) class can be used to search strings. These searches can range in complexity from simple to complicated text patterns.

The following code example searches for the word "the" or "their" in a sentence, ignoring case. The static method [Regex.IsMatch](#) performs the search. You give it the string to search and a search pattern. In this case, a third argument specifies case-insensitive search. For more information, see [System.Text.RegularExpressions.RegexOptions](#).

The search pattern describes the text you search for. The following table describes each element of the search pattern. (The table below uses the single `\`, which must be escaped as `\\` in a C# string).

PATTERN	MEANING
<code>the</code>	match the text "the"
<code>(eir)?</code>	match 0 or 1 occurrence of "eir"
<code>\s</code>	match a white-space character

```

string[] sentences =
{
    "Put the water over there.",
    "They're quite thirsty.",
    "Their water bottles broke."
};

string sPattern = "the(ir)?\\s";

foreach (string s in sentences)
{
    Console.Write($"{s,24}");

    if (System.Text.RegularExpressions.Regex.IsMatch(s, sPattern,
System.Text.RegularExpressions.RegexOptions.IgnoreCase))
    {
        Console.WriteLine($" (match for '{sPattern}' found)");
    }
    else
    {
        Console.WriteLine();
    }
}

```

TIP

The `string` methods are usually better choices when you are searching for an exact string. Regular expressions are better when you are searching for some pattern in a source string.

Does a string follow a pattern?

The following code uses regular expressions to validate the format of each string in an array. The validation requires that each string have the form of a telephone number in which three groups of digits are separated by dashes, the first two groups contain three digits, and the third group contains four digits. The search pattern uses the regular expression `^\d{3}-\d{3}-\d{4}$`. For more information, see [Regular Expression Language - Quick Reference](#).

PATTERN	MEANING
<code>^</code>	matches the beginning of the string
<code>\d{3}</code>	matches exactly 3 digit characters
<code>-</code>	matches the '-' character
<code>\d{4}</code>	matches exactly 4 digit characters
<code>\$</code>	matches the end of the string

```
string[] numbers =
{
    "123-555-0190",
    "444-234-22450",
    "690-555-0178",
    "146-893-232",
    "146-555-0122",
    "4007-555-0111",
    "407-555-0111",
    "407-2-5555",
    "407-555-8974",
    "407-2ab-5555",
    "690-555-8148",
    "146-893-232-"
};

string sPattern = "^\d{3}-\d{3}-\d{4}$";

foreach (string s in numbers)
{
    Console.WriteLine($"{s,14}");

    if (System.Text.RegularExpressions.Regex.IsMatch(s, sPattern))
    {
        Console.WriteLine(" - valid");
    }
    else
    {
        Console.WriteLine(" - invalid");
    }
}
```

This single search pattern matches many valid strings. Regular expressions are better to search for or validate

against a pattern, rather than a single text string.

See also

- [C# programming guide](#)
- [Strings](#)
- [LINQ and strings](#)
- [System.Text.RegularExpressions.Regex](#)
- [.NET regular expressions](#)
- [Regular expression language - quick reference](#)
- [Best practices for using strings in .NET](#)

How to modify string contents in C#

12/28/2021 • 5 minutes to read • [Edit Online](#)

This article demonstrates several techniques to produce a `string` by modifying an existing `string`. All the techniques demonstrated return the result of the modifications as a new `string` object. To demonstrate that the original and modified strings are distinct instances, the examples store the result in a new variable. You can examine the original `string` and the new, modified `string` when you run each example.

NOTE

The C# examples in this article run in the [Try.NET](#) inline code runner and playground. Select the **Run** button to run an example in an interactive window. Once you execute the code, you can modify it and run the modified code by selecting **Run** again. The modified code either runs in the interactive window or, if compilation fails, the interactive window displays all C# compiler error messages.

There are several techniques demonstrated in this article. You can replace existing text. You can search for patterns and replace matching text with other text. You can treat a string as a sequence of characters. You can also use convenience methods that remove white space. Choose the techniques that most closely match your scenario.

Replace text

The following code creates a new string by replacing existing text with a substitute.

```
string source = "The mountains are behind the clouds today.";

// Replace one substring with another with String.Replace.
// Only exact matches are supported.
var replacement = source.Replace("mountains", "peaks");
Console.WriteLine($"The source string is <{source}>");
Console.WriteLine($"The updated string is <{replacement}>");
```

The preceding code demonstrates this *immutable* property of strings. You can see in the preceding example that the original string, `source`, is not modified. The [String.Replace](#) method creates a new `string` containing the modifications.

The [Replace](#) method can replace either strings or single characters. In both cases, every occurrence of the sought text is replaced. The following example replaces all ' ' characters with '_':

```
string source = "The mountains are behind the clouds today.";

// Replace all occurrences of one char with another.
var replacement = source.Replace(' ', '_');
Console.WriteLine(source);
Console.WriteLine(replacement);
```

The source string is unchanged, and a new string is returned with the replacement.

Trim white space

You can use the [String.Trim](#), [String.TrimStart](#), and [String.TrimEnd](#) methods to remove any leading or trailing white

space. The following code shows an example of each. The source string does not change; these methods return a new string with the modified contents.

```
// Remove trailing and leading white space.
string source = "    I'm wider than I need to be.    ";
// Store the results in a new string variable.
var trimmedResult = source.Trim();
var trimLeading = source.TrimStart();
var trimTrailing = source.TrimEnd();
Console.WriteLine($"<{source}>");
Console.WriteLine($"<{trimmedResult}>");
Console.WriteLine($"<{trimLeading}>");
Console.WriteLine($"<{trimTrailing}>");
```

Remove text

You can remove text from a string using the [String.Remove](#) method. This method removes a number of characters starting at a specific index. The following example shows how to use [String.IndexOf](#) followed by [Remove](#) to remove text from a string:

```
string source = "Many mountains are behind many clouds today.";
// Remove a substring from the middle of the string.
string toRemove = "many ";
string result = string.Empty;
int i = source.IndexOf(toRemove);
if (i >= 0)
{
    result= source.Remove(i, toRemove.Length);
}
Console.WriteLine(source);
Console.WriteLine(result);
```

Replace matching patterns

You can use [regular expressions](#) to replace text matching patterns with new text, possibly defined by a pattern. The following example uses the [System.Text.RegularExpressions.Regex](#) class to find a pattern in a source string and replace it with proper capitalization. The [Regex.Replace\(String, String, MatchEvaluator, RegexOptions\)](#) method takes a function that provides the logic of the replacement as one of its arguments. In this example, that function, `LocalReplaceMatchCase` is a **local function** declared inside the sample method. `LocalReplaceMatchCase` uses the [System.Text.StringBuilder](#) class to build the replacement string with proper capitalization.

Regular expressions are most useful for searching and replacing text that follows a pattern, rather than known text. For more information, see [How to search strings](#). The search pattern, "the\s" searches for the word "the" followed by a white-space character. That part of the pattern ensures that it doesn't match "there" in the source string. For more information on regular expression language elements, see [Regular Expression Language - Quick Reference](#).


```

string source = "The mountains are still there behind the clouds today.";

// Use Regex.Replace for more flexibility.
// Replace "the" or "The" with "many" or "Many".
// using System.Text.RegularExpressions
string replaceWith = "many ";
source = System.Text.RegularExpressions.Regex.Replace(source, "the\\s", LocalReplaceMatchCase,
    System.Text.RegularExpressions.RegexOptions.IgnoreCase);
Console.WriteLine(source);

string LocalReplaceMatchCase(System.Text.RegularExpressions.Match matchExpression)
{
    // Test whether the match is capitalized
    if (Char.IsUpper(matchExpression.Value[0]))
    {
        // Capitalize the replacement string
        System.Text.StringBuilder replacementBuilder = new System.Text.StringBuilder(replaceWith);
        replacementBuilder[0] = Char.ToUpper(replacementBuilder[0]);
        return replacementBuilder.ToString();
    }
    else
    {
        return replaceWith;
    }
}

```

The [StringBuilder.ToString](#) method returns an immutable string with the contents in the [StringBuilder](#) object.

Modifying individual characters

You can produce a character array from a string, modify the contents of the array, and then create a new string from the modified contents of the array.

The following example shows how to replace a set of characters in a string. First, it uses the [String.ToCharArray\(\)](#) method to create an array of characters. It uses the [IndexOf](#) method to find the starting index of the word "fox." The next three characters are replaced with a different word. Finally, a new string is constructed from the updated character array.

```

string phrase = "The quick brown fox jumps over the fence";
Console.WriteLine(phrase);

char[] phraseAsChars = phrase.ToCharArray();
int animalIndex = phrase.IndexOf("fox");
if (animalIndex != -1)
{
    phraseAsChars[animalIndex++] = 'c';
    phraseAsChars[animalIndex++] = 'a';
    phraseAsChars[animalIndex] = 't';
}

string updatedPhrase = new string(phraseAsChars);
Console.WriteLine(updatedPhrase);

```

Programmatically build up string content

Since strings are immutable, the previous examples all create temporary strings or character arrays. In high-performance scenarios, it may be desirable to avoid these heap allocations. .NET Core provides a [String.Create](#) method that allows you to programmatically fill in the character content of a string via a callback while avoiding the intermediate temporary string allocations.

```
// constructing a string from a char array, prefix it with some additional characters
char[] chars = { 'a', 'b', 'c', 'd', '\0' };
int length = chars.Length + 2;
string result = string.Create(length, chars, (Span<char> strContent, char[] charArray) =>
{
    strContent[0] = '0';
    strContent[1] = '1';
    for (int i = 0; i < charArray.Length; i++)
    {
        strContent[i + 2] = charArray[i];
    }
});

Console.WriteLine(result);
```

You could modify a string in a fixed block with unsafe code, but it is **strongly** discouraged to modify the string content after a string is created. Doing so will break things in unpredictable ways. For example, if someone interns a string that has the same content as yours, they'll get your copy and won't expect that you are modifying their string.

See also

- [.NET regular expressions](#)
- [Regular expression language - quick reference](#)

How to compare strings in C#

12/28/2021 • 11 minutes to read • [Edit Online](#)

You compare strings to answer one of two questions: "Are these two strings equal?" or "In what order should these strings be placed when sorting them?"

Those two questions are complicated by factors that affect string comparisons:

- You can choose an ordinal or linguistic comparison.
- You can choose if case matters.
- You can choose culture-specific comparisons.
- Linguistic comparisons are culture and platform-dependent.

NOTE

The C# examples in this article run in the [Try.NET](#) inline code runner and playground. Select the **Run** button to run an example in an interactive window. Once you execute the code, you can modify it and run the modified code by selecting **Run** again. The modified code either runs in the interactive window or, if compilation fails, the interactive window displays all C# compiler error messages.

When you compare strings, you define an order among them. Comparisons are used to sort a sequence of strings. Once the sequence is in a known order, it is easier to search, both for software and for humans. Other comparisons may check if strings are the same. These sameness checks are similar to equality, but some differences, such as case differences, may be ignored.

Default ordinal comparisons

By default, the most common operations:

- [String.Equals](#)
- [String.Equality](#) and [String.Inequality](#), that is, [equality operators](#) `==` and `!=`, respectively

perform a case-sensitive ordinal comparison, and in the case of [String.Equals](#) a [StringComparison](#) argument can be provided to alter its sorting rules. The following example demonstrates that:

```
string root = @"C:\users";
string root2 = @"C:\Users";

bool result = root.Equals(root2);
Console.WriteLine($"Ordinal comparison: <{root}> and <{root2}> are {(result ? "equal." : "not equal.")}");

result = root.Equals(root2, StringComparison.Ordinal);
Console.WriteLine($"Ordinal comparison: <{root}> and <{root2}> are {(result ? "equal." : "not equal.")}");

Console.WriteLine($"Using == says that <{root}> and <{root2}> are {(root == root2 ? "equal" : "not equal.")}");
```

The default ordinal comparison doesn't take linguistic rules into account when comparing strings. It compares the binary value of each [Char](#) object in two strings. As a result, the default ordinal comparison is also case-sensitive.

The test for equality with [String.Equals](#) and the `==` and `!=` operators differs from string comparison using the

[String.CompareTo](#) and [Compare\(String, String\)](#) methods. While the tests for equality perform a case-sensitive ordinal comparison, the comparison methods perform a case-sensitive, culture-sensitive comparison using the current culture. Because the default comparison methods often perform different types of comparisons, we recommend that you always make the intent of your code clear by calling an overload that explicitly specifies the type of comparison to perform.

Case-insensitive ordinal comparisons

The [String.Equals\(String, StringComparison\)](#) method enables you to specify a [StringComparison](#) value of [StringComparison.OrdinalIgnoreCase](#) for a case-insensitive ordinal comparison. There is also a static [String.Compare\(String, String, StringComparison\)](#) method that performs a case-insensitive ordinal comparison if you specify a value of [StringComparison.OrdinalIgnoreCase](#) for the [StringComparison](#) argument. These are shown in the following code:

```
string root = @"C:\users";
string root2 = @"C:\Users";

bool result = root.Equals(root2, StringComparison.OrdinalIgnoreCase);
bool areEqual = String.Equals(root, root2, StringComparison.OrdinalIgnoreCase);
int comparison = String.Compare(root, root2, comparisonType: StringComparison.OrdinalIgnoreCase);

Console.WriteLine($"Ordinal ignore case: <{root}> and <{root2}> are {(result ? "equal." : "not equal.")}");
Console.WriteLine($"Ordinal static ignore case: <{root}> and <{root2}> are {(areEqual ? "equal." : "not equal.")}");
if (comparison < 0)
    Console.WriteLine($"<{root}> is less than <{root2}>");
else if (comparison > 0)
    Console.WriteLine($"<{root}> is greater than <{root2}>");
else
    Console.WriteLine($"<{root}> and <{root2}> are equivalent in order");
```

When performing a case-insensitive ordinal comparison, these methods use the casing conventions of the [invariant culture](#).

Linguistic comparisons

Strings can also be ordered using linguistic rules for the current culture. This is sometimes referred to as "word sort order." When you perform a linguistic comparison, some nonalphanumeric Unicode characters might have special weights assigned. For example, the hyphen "-" may have a small weight assigned to it so that "co-op" and "coop" appear next to each other in sort order. In addition, some Unicode characters may be equivalent to a sequence of [Char](#) instances. The following example uses the phrase "They dance in the street." in German with the "ss" (U+0073 U+0073) in one string and 'ß' (U+00DF) in another. Linguistically (in Windows), "ss" is equal to the German Esszet: 'ß' character in both the "en-US" and "de-DE" cultures.

```

string first = "Sie tanzen auf der Straße.";
string second = "Sie tanzen auf der Strasse.";

Console.WriteLine($"First sentence is <{first}>");
Console.WriteLine($"Second sentence is <{second}>");

bool equal = String.Equals(first, second, StringComparison.InvariantCulture);
Console.WriteLine($"The two strings {(equal == true ? "are" : "are not")} equal.");
showComparison(first, second);

string word = "coop";
string words = "co-op";
string other = "cop";

showComparison(word, words);
showComparison(word, other);
showComparison(words, other);
void showComparison(string one, string two)
{
    int compareLinguistic = String.Compare(one, two, StringComparison.InvariantCulture);
    int compareOrdinal = String.Compare(one, two, StringComparison.Ordinal);
    if (compareLinguistic < 0)
        Console.WriteLine($"<{one}> is less than <{two}> using invariant culture");
    else if (compareLinguistic > 0)
        Console.WriteLine($"<{one}> is greater than <{two}> using invariant culture");
    else
        Console.WriteLine($"<{one}> and <{two}> are equivalent in order using invariant culture");
    if (compareOrdinal < 0)
        Console.WriteLine($"<{one}> is less than <{two}> using ordinal comparison");
    else if (compareOrdinal > 0)
        Console.WriteLine($"<{one}> is greater than <{two}> using ordinal comparison");
    else
        Console.WriteLine($"<{one}> and <{two}> are equivalent in order using ordinal comparison");
}

```

This sample demonstrates the operating system-dependent nature of linguistic comparisons. The host for the interactive window is a Linux host. The linguistic and ordinal comparisons produce the same results. If you run this same sample on a Windows host, you see the following output:

```

<coop> is less than <co-op> using invariant culture
<coop> is greater than <co-op> using ordinal comparison
<coop> is less than <cop> using invariant culture
<coop> is less than <cop> using ordinal comparison
<co-op> is less than <cop> using invariant culture
<co-op> is less than <cop> using ordinal comparison

```

On Windows, the sort order of "cop", "coop", and "co-op" change when you change from a linguistic comparison to an ordinal comparison. The two German sentences also compare differently using the different comparison types.

Comparisons using specific cultures

This sample stores [CultureInfo](#) objects for the en-US and de-DE cultures. The comparisons are performed using a [CultureInfo](#) object to ensure a culture-specific comparison.

The culture used affects linguistic comparisons. The following example shows the results of comparing the two German sentences using the "en-US" culture and the "de-DE" culture:

```

string first = "Sie tanzen auf der Straße.";
string second = "Sie tanzen auf der Strasse.";

Console.WriteLine($"First sentence is <{first}>");
Console.WriteLine($"Second sentence is <{second}>");

var en = new System.Globalization.CultureInfo("en-US");

// For culture-sensitive comparisons, use the String.Compare
// overload that takes a StringComparison value.
int i = String.Compare(first, second, en, System.Globalization.CompareOptions.None);
Console.WriteLine($"Comparing in {en.Name} returns {i}.");

var de = new System.Globalization.CultureInfo("de-DE");
i = String.Compare(first, second, de, System.Globalization.CompareOptions.None);
Console.WriteLine($"Comparing in {de.Name} returns {i}.");

bool b = String.Equals(first, second, StringComparison.CurrentCulture);
Console.WriteLine($"The two strings {(b ? "are" : "are not")} equal.");

string word = "coop";
string words = "co-op";
string other = "cop";

showComparison(word, words, en);
showComparison(word, other, en);
showComparison(words, other, en);
void showComparison(string one, string two, System.Globalization.CultureInfo culture)
{
    int compareLinguistic = String.Compare(one, two, en, System.Globalization.CompareOptions.None);
    int compareOrdinal = String.Compare(one, two, StringComparison.Ordinal);
    if (compareLinguistic < 0)
        Console.WriteLine($"<{one}> is less than <{two}> using en-US culture");
    else if (compareLinguistic > 0)
        Console.WriteLine($"<{one}> is greater than <{two}> using en-US culture");
    else
        Console.WriteLine($"<{one}> and <{two}> are equivalent in order using en-US culture");
    if (compareOrdinal < 0)
        Console.WriteLine($"<{one}> is less than <{two}> using ordinal comparison");
    else if (compareOrdinal > 0)
        Console.WriteLine($"<{one}> is greater than <{two}> using ordinal comparison");
    else
        Console.WriteLine($"<{one}> and <{two}> are equivalent in order using ordinal comparison");
}

```

Culture-sensitive comparisons are typically used to compare and sort strings input by users with other strings input by users. The characters and sorting conventions of these strings might vary depending on the locale of the user's computer. Even strings that contain identical characters might sort differently depending on the culture of the current thread. In addition, try this sample code locally on a Windows machine, and you'll get the following results:

```

<coop> is less than <co-op> using en-US culture
<coop> is greater than <co-op> using ordinal comparison
<coop> is less than <cop> using en-US culture
<coop> is less than <cop> using ordinal comparison
<co-op> is less than <cop> using en-US culture
<co-op> is less than <cop> using ordinal comparison

```

Linguistic comparisons are dependent on the current culture, and are OS dependent. Take that into account when you work with string comparisons.

Linguistic sorting and searching strings in arrays

The following examples show how to sort and search for strings in an array using a linguistic comparison dependent on the current culture. You use the static [Array](#) methods that take a [System.StringComparer](#) parameter.

This example shows how to sort an array of strings using the current culture:

```
string[] lines = new string[]
{
    @"c:\public\textfile.txt",
    @"c:\public\textFile.TXT",
    @"c:\public\Text.txt",
    @"c:\public\testfile2.txt"
};

Console.WriteLine("Non-sorted order:");
foreach (string s in lines)
{
    Console.WriteLine($"    {s}");
}

Console.WriteLine("\n\rSorted order:");

// Specify Ordinal to demonstrate the different behavior.
Array.Sort(lines, StringComparer.CurrentCulture);

foreach (string s in lines)
{
    Console.WriteLine($"    {s}");
}
```

Once the array is sorted, you can search for entries using a binary search. A binary search starts in the middle of the collection to determine which half of the collection would contain the sought string. Each subsequent comparison subdivides the remaining part of the collection in half. The array is sorted using the [StringComparer.CurrentCulture](#). The local function `ShowWhere` displays information about where the string was found. If the string wasn't found, the returned value indicates where it would be if it were found.

```

string[] lines = new string[]
{
    @"c:\public\textfile.txt",
    @"c:\public\textFile.TXT",
    @"c:\public\Text.txt",
    @"c:\public\testfile2.txt"
};
Array.Sort(lines, StringComparer.CurrentCulture);

string searchString = @"c:\public\TEXTFILE.TXT";
Console.WriteLine($"Binary search for <{searchString}>");
int result = Array.BinarySearch(lines, searchString, StringComparer.CurrentCulture);
ShowWhere<string>(lines, result);

Console.WriteLine($"{{(result > 0 ? "Found" : "Did not find")}} {searchString}");

void ShowWhere<T>(T[] array, int index)
{
    if (index < 0)
    {
        index = ~index;

        Console.Write("Not found. Sorts between: ");

        if (index == 0)
            Console.Write("beginning of sequence and ");
        else
            Console.Write($"{{array[index - 1]}} and ");

        if (index == array.Length)
            Console.WriteLine("end of sequence.");
        else
            Console.WriteLine($"{{array[index]}}.");
    }
    else
    {
        Console.WriteLine($"Found at index {index}.");
    }
}

```

Ordinal sorting and searching in collections

The following code uses the [System.Collections.Generic.List<T>](#) collection class to store strings. The strings are sorted using the [List<T>.Sort](#) method. This method needs a delegate that compares and orders two strings. The [String.CompareTo](#) method provides that comparison function. Run the sample and observe the order. This sort operation uses an ordinal case-sensitive sort. You would use the static [String.Compare](#) methods to specify different comparison rules.


```

List<string> lines = new List<string>
{
    @"c:\public\textfile.txt",
    @"c:\public\textFile.TXT",
    @"c:\public\Text.txt",
    @"c:\public\testfile2.txt"
};

Console.WriteLine("Non-sorted order:");
foreach (string s in lines)
{
    Console.WriteLine($"    {s}");
}

Console.WriteLine("\n\rSorted order:");

lines.Sort((left, right) => left.CompareTo(right));
foreach (string s in lines)
{
    Console.WriteLine($"    {s}");
}

```

Once sorted, the list of strings can be searched using a binary search. The following sample shows how to search the sorted list using the same comparison function. The local function `ShowWhere` shows where the sought text is or would be:

```

List<string> lines = new List<string>
{
    @"c:\public\textfile.txt",
    @"c:\public\textFile.TXT",
    @"c:\public\Text.txt",
    @"c:\public\testfile2.txt"
};
lines.Sort((left, right) => left.CompareTo(right));

string searchString = @"c:\public\TEXTFILE.TXT";
Console.WriteLine($"Binary search for <{searchString}>");
int result = lines.BinarySearch(searchString);
ShowWhere<string>(lines, result);

Console.WriteLine($"{(result > 0 ? "Found" : "Did not find")} {searchString}");

void ShowWhere<T>(IList<T> collection, int index)
{
    if (index < 0)
    {
        index = ~index;

        Console.Write("Not found. Sorts between: ");

        if (index == 0)
            Console.Write("beginning of sequence and ");
        else
            Console.Write($"{collection[index - 1]} and ");

        if (index == collection.Count)
            Console.WriteLine("end of sequence.");
        else
            Console.WriteLine($"{collection[index]}.");
    }
    else
    {
        Console.WriteLine($"Found at index {index}.");
    }
}

```

Always make sure to use the same type of comparison for sorting and searching. Using different comparison types for sorting and searching produces unexpected results.

Collection classes such as [System.Collections.Hashtable](#), [System.Collections.Generic.Dictionary<TKey,TValue>](#), and [System.Collections.Generic.List<T>](#) have constructors that take a [System.StringComparer](#) parameter when the type of the elements or keys is `string`. In general, you should use these constructors whenever possible, and specify either [StringComparer.Ordinal](#) or [StringComparer.OrdinalIgnoreCase](#).

Reference equality and string interning

None of the samples have used [ReferenceEquals](#). This method determines if two strings are the same object, which can lead to inconsistent results in string comparisons. The following example demonstrates the *string interning* feature of C#. When a program declares two or more identical string variables, the compiler stores them all in the same location. By calling the [ReferenceEquals](#) method, you can see that the two strings actually refer to the same object in memory. Use the [String.Copy](#) method to avoid interning. After the copy has been made, the two strings have different storage locations, even though they have the same value. Run the following sample to show that strings `a` and `b` are *interned* meaning they share the same storage. The strings `a` and `c` are not.

```
string a = "The computer ate my source code.";
string b = "The computer ate my source code.";

if (String.ReferenceEquals(a, b))
    Console.WriteLine("a and b are interned.");
else
    Console.WriteLine("a and b are not interned.");

string c = String.Copy(a);

if (String.ReferenceEquals(a, c))
    Console.WriteLine("a and c are interned.");
else
    Console.WriteLine("a and c are not interned.");
```

NOTE

When you test for equality of strings, you should use the methods that explicitly specify what kind of comparison you intend to perform. Your code is much more maintainable and readable. Use the overloads of the methods of the [System.String](#) and [System.Array](#) classes that take a [StringComparison](#) enumeration parameter. You specify which type of comparison to perform. Avoid using the `==` and `!=` operators when you test for equality. The [String.CompareTo](#) instance methods always perform an ordinal case-sensitive comparison. They are primarily suited for ordering strings alphabetically.

You can intern a string or retrieve a reference to an existing interned string by calling the [String.Intern](#) method. To determine whether a string is interned, call the [String.IsInterned](#) method.

See also

- [System.Globalization.CultureInfo](#)
- [System.StringComparer](#)
- [Strings](#)
- [Comparing strings](#)
- [Globalizing and localizing applications](#)

How to catch a non-CLS exception

12/28/2021 • 2 minutes to read • [Edit Online](#)

Some .NET languages, including C++/CLI, allow objects to throw exceptions that do not derive from [Exception](#). Such exceptions are called *non-CLS exceptions* or *non-Exceptions*. In C# you cannot throw non-CLS exceptions, but you can catch them in two ways:

- Within a `catch (RuntimeWrappedException e)` block.

By default, a Visual C# assembly catches non-CLS exceptions as wrapped exceptions. Use this method if you need access to the original exception, which can be accessed through the [RuntimeWrappedException.WrappedException](#) property. The procedure later in this topic explains how to catch exceptions in this manner.

- Within a general catch block (a catch block without an exception type specified) that is put after all other `catch` blocks.

Use this method when you want to perform some action (such as writing to a log file) in response to non-CLS exceptions, and you do not need access to the exception information. By default the common language runtime wraps all exceptions. To disable this behavior, add this assembly-level attribute to your code, typically in the AssemblyInfo.cs file:

```
[assembly: RuntimeCompatibilityAttribute(WrapNonExceptionThrows = false)]
```

To catch a non-CLS exception

Within a `catch(RuntimeWrappedException e)` block, access the original exception through the [RuntimeWrappedException.WrappedException](#) property.

Example

The following example shows how to catch a non-CLS exception that was thrown from a class library written in C++/CLI. Note that in this example, the C# client code knows in advance that the exception type being thrown is a [System.String](#). You can cast the [RuntimeWrappedException.WrappedException](#) property back its original type as long as that type is accessible from your code.

```
// Class library written in C++/CLI.
var myClass = new ThrowNonCLS.Class1();

try
{
    // throws gcnew System::String(
    // "I do not derive from System.Exception!");
    myClass.TestThrow();
}
catch (RuntimeWrappedException e)
{
    String s = e.WrappedException as String;
    if (s != null)
    {
        Console.WriteLine(s);
    }
}
```

See also

- [RuntimeWrappedException](#)
- [Exceptions and Exception Handling](#)

The .NET Compiler Platform SDK

12/28/2021 • 6 minutes to read • [Edit Online](#)

Compilers build a detailed model of application code as they validate the syntax and semantics of that code. They use this model to build the executable output from the source code. The .NET Compiler Platform SDK provides access to this model. Increasingly, we rely on integrated development environment (IDE) features such as IntelliSense, refactoring, intelligent rename, "Find all references," and "Go to definition" to increase our productivity. We rely on code analysis tools to improve our code quality, and code generators to aid in application construction. As these tools get smarter, they need access to more and more of the model that only compilers create as they process application code. This is the core mission of the Roslyn APIs: opening up the opaque boxes and allowing tools and end users to share in the wealth of information compilers have about our code. Instead of being opaque source-code-in and object-code-out translators, through Roslyn, compilers become platforms: APIs that you can use for code-related tasks in your tools and applications.

.NET Compiler Platform SDK concepts

The .NET Compiler Platform SDK dramatically lowers the barrier to entry for creating code focused tools and applications. It creates many opportunities for innovation in areas such as meta-programming, code generation and transformation, interactive use of the C# and Visual Basic languages, and embedding of C# and Visual Basic in domain-specific languages.

The .NET Compiler Platform SDK enables you to build *analyzers* and *code fixes* that find and correct coding mistakes. *Analyzers* understand the syntax (structure of code) and semantics to detect practices that should be corrected. *Code fixes* provide one or more suggested fixes for addressing coding mistakes found by analyzers or compiler diagnostics. Typically, an analyzer and the associated code fixes are packaged together in a single project.

Analyzers and code fixes use static analysis to understand code. They do not run the code or provide other testing benefits. They can, however, point out practices that often lead to bugs, unmaintainable code, or standard guideline violation.

In addition to analyzers and code fixes, The .NET Compiler Platform SDK also enables you to build *code refactorings*. It also provides a single set of APIs that enable you to examine and understand a C# or Visual Basic codebase. Because you can use this single codebase, you can write analyzers and code fixes more easily by leveraging the syntactic and semantic analysis APIs provided by the .NET Compiler Platform SDK. Freed from the large task of replicating the analysis done by the compiler, you can concentrate on the more focused task of finding and fixing common coding errors for your project or library.

A smaller benefit is that your analyzers and code fixes are smaller and use much less memory when loaded in Visual Studio than they would if you wrote your own codebase to understand the code in a project. By leveraging the same classes used by the compiler and Visual Studio, you can create your own static analysis tools. This means your team can use analyzers and code fixes without a noticeable impact on the IDE's performance.

There are three main scenarios for writing analyzers and code fixes:

1. [Enforce team coding standards](#)
2. [Provide guidance with library packages](#)
3. [Provide general guidance](#)

Enforce team coding standards

Many teams have coding standards that are enforced through code reviews with other team members. Analyzers and code fixes can make this process much more efficient. Code reviews happen when a developer shares their work with others on the team. The developer will have invested all the time needed to complete a new feature before getting any comments. Weeks may go by while the developer reinforces habits that don't match the team's practices.

Analyzers run as a developer writes code. The developer gets immediate feedback that encourages following the guidance immediately. The developer builds habits to write compliant code as soon as they begin prototyping. When the feature is ready for humans to review, all the standard guidance has been enforced.

Teams can build analyzers and code fixes that look for the most common practices that violate team coding practices. These can be installed on each developer's machine to enforce the standards.

TIP

Before building your own analyzer, check out the built-in ones. For more information, see [Code-style rules](#).

Provide guidance with library packages

There is a wealth of libraries available for .NET developers on NuGet. Some of these come from Microsoft, some from third-party companies, and others from community members and volunteers. These libraries get more adoption and higher reviews when developers can succeed with those libraries.

In addition to providing documentation, you can provide analyzers and code fixes that find and correct common mis-uses of your library. These immediate corrections will help developers succeed more quickly.

You can package analyzers and code fixes with your library on NuGet. In that scenario, every developer who installs your NuGet package will also install the analyzer package. All developers using your library will immediately get guidance from your team in the form of immediate feedback on mistakes and suggested corrections.

Provide general guidance

The .NET developer community has discovered, through experience, patterns that work well and patterns that are best avoided. Several community members have created analyzers that enforce those recommended patterns. As we learn more, there is always room for new ideas.

These analyzers can be uploaded to the [Visual Studio Marketplace](#) and downloaded by developers using Visual Studio. Newcomers to the language and the platform learn accepted practices quickly and become productive earlier in their .NET journey. As these become more widely used, the community adopts these practices.

Next steps

The .NET Compiler Platform SDK includes the latest language object models for code generation, analysis, and refactoring. This section provides a conceptual overview of the .NET Compiler Platform SDK. Further details can be found in the quickstarts, samples, and tutorials sections.

You can learn more about the concepts in the .NET Compiler Platform SDK in these five topics:

- [Explore code with the syntax visualizer](#)
- [Understand the compiler API model](#)
- [Work with syntax](#)
- [Work with semantics](#)
- [Work with a workspace](#)

To get started, you'll need to install the **.NET Compiler Platform SDK**:

Installation instructions - Visual Studio Installer

There are two different ways to find the **.NET Compiler Platform SDK** in the **Visual Studio Installer**:

Install using the Visual Studio Installer - Workloads view

The **.NET Compiler Platform SDK** is not automatically selected as part of the Visual Studio extension development workload. You must select it as an optional component.

1. Run **Visual Studio Installer**
2. Select **Modify**
3. Check the **Visual Studio extension development** workload.
4. Open the **Visual Studio extension development** node in the summary tree.
5. Check the box for **.NET Compiler Platform SDK**. You'll find it last under the optional components.

Optionally, you'll also want the **DGML editor** to display graphs in the visualizer:

1. Open the **Individual components** node in the summary tree.
2. Check the box for **DGML editor**

Install using the Visual Studio Installer - Individual components tab

1. Run **Visual Studio Installer**
2. Select **Modify**
3. Select the **Individual components** tab
4. Check the box for **.NET Compiler Platform SDK**. You'll find it at the top under the **Compilers, build tools, and runtimes** section.

Optionally, you'll also want the **DGML editor** to display graphs in the visualizer:

1. Check the box for **DGML editor**. You'll find it under the **Code tools** section.

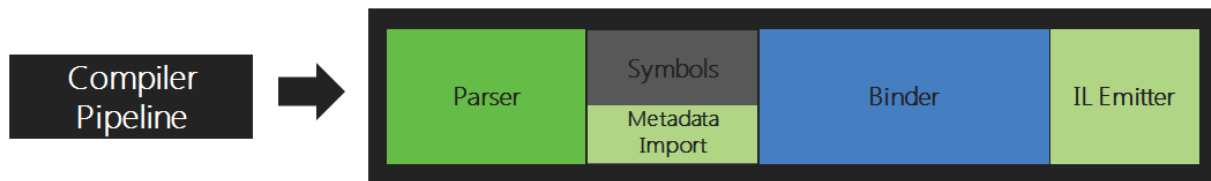
Understand the .NET Compiler Platform SDK model

12/28/2021 • 3 minutes to read • [Edit Online](#)

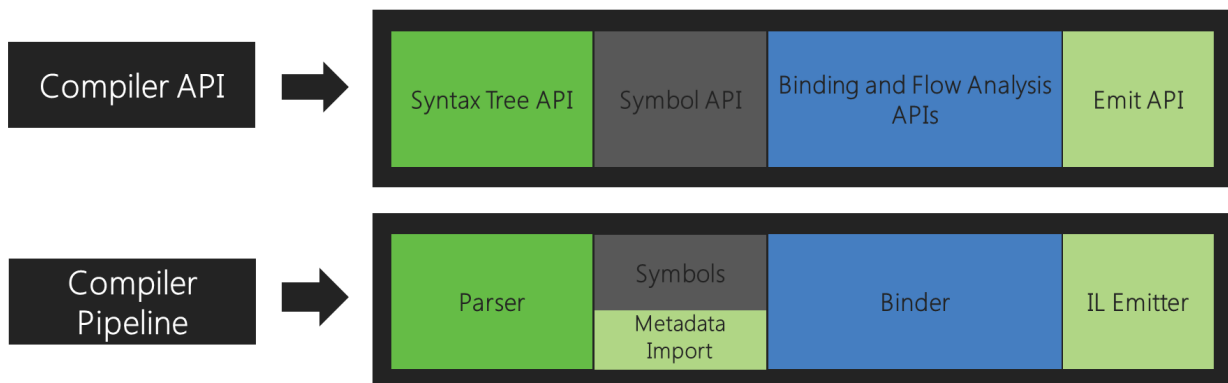
Compilers process the code you write following structured rules that often differ from the way humans read and understand code. A basic understanding of the model used by compilers is essential to understanding the APIs you use when building Roslyn-based tools.

Compiler pipeline functional areas

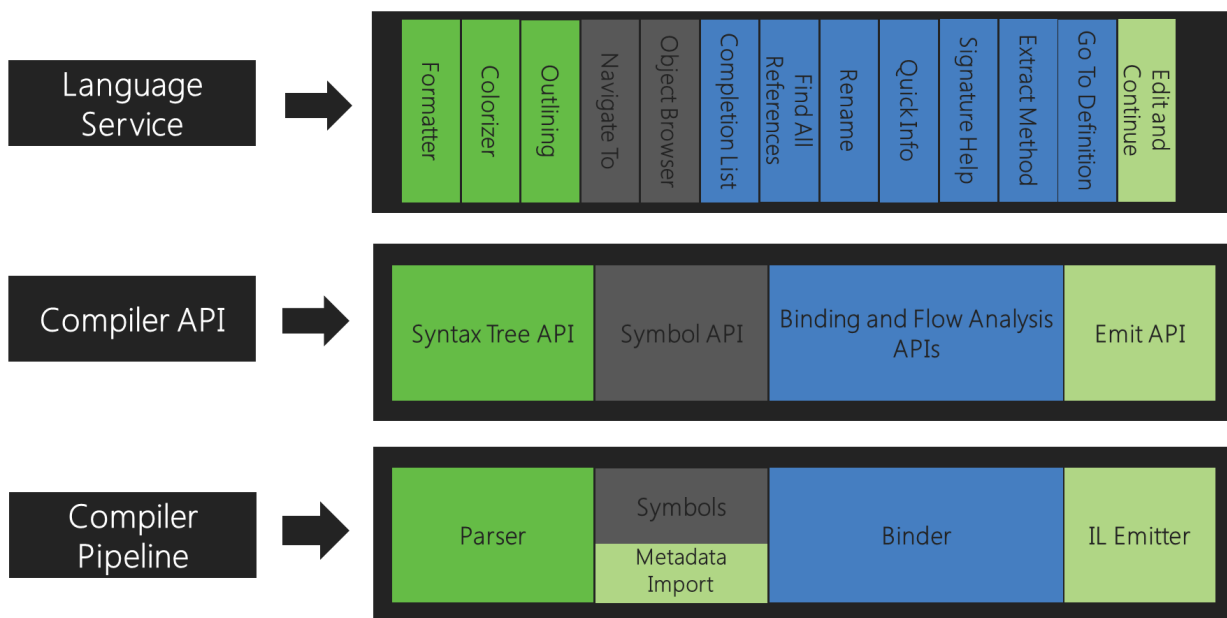
The .NET Compiler Platform SDK exposes the C# and Visual Basic compilers' code analysis to you as a consumer by providing an API layer that mirrors a traditional compiler pipeline.



Each phase of this pipeline is a separate component. First, the parse phase tokenizes and parses source text into syntax that follows the language grammar. Second, the declaration phase analyzes source and imported metadata to form named symbols. Next, the bind phase matches identifiers in the code to symbols. Finally, the emit phase emits an assembly with all the information built up by the compiler.



Corresponding to each of those phases, the .NET Compiler Platform SDK exposes an object model that allows access to the information at that phase. The parsing phase exposes a syntax tree, the declaration phase exposes a hierarchical symbol table, the binding phase exposes the result of the compiler's semantic analysis, and the emit phase is an API that produces IL byte codes.



Each compiler combines these components together as a single end-to-end whole.

These APIs are the same ones used by Visual Studio. For instance, the code outlining and formatting features use the syntax trees, the **Object Browser**, and navigation features use the symbol table, refactorings and **Go to Definition** use the semantic model, and **Edit and Continue** uses all of these, including the Emit API.

API layers

The .NET compiler SDK consists of several layers of APIs: compiler APIs, diagnostic APIs, scripting APIs, and workspaces APIs.

Compiler APIs

The compiler layer contains the object models that correspond to information exposed at each phase of the compiler pipeline, both syntactic and semantic. The compiler layer also contains an immutable snapshot of a single invocation of a compiler, including assembly references, compiler options, and source code files. There are two distinct APIs that represent the C# language and the Visual Basic language. The two APIs are similar in shape but tailored for high-fidelity to each individual language. This layer has no dependencies on Visual Studio components.

Diagnostic APIs

As part of its analysis, the compiler may produce a set of diagnostics covering everything from syntax, semantic, and definite assignment errors to various warnings and informational diagnostics. The Compiler API layer exposes diagnostics through an extensible API that allows user-defined analyzers to be plugged into the compilation process. It allows user-defined diagnostics, such as those produced by tools like StyleCop, to be produced alongside compiler-defined diagnostics. Producing diagnostics in this way has the benefit of integrating naturally with tools such as MSBuild and Visual Studio, which depend on diagnostics for experiences such as halting a build based on policy and showing live squiggles in the editor and suggesting code fixes.

Scripting APIs

Hosting and scripting APIs are part of the compiler layer. You can use them for executing code snippets and accumulating a runtime execution context. The C# interactive REPL (Read-Evaluate-Print Loop) uses these APIs. The REPL enables you to use C# as a scripting language, executing the code interactively as you write it.

Workspaces APIs

The Workspaces layer contains the Workspace API, which is the starting point for doing code analysis and refactoring over entire solutions. It assists you in organizing all the information about the projects in a solution into a single object model, offering you direct access to the compiler layer object models without needing to

parse files, configure options, or manage project-to-project dependencies.

In addition, the Workspaces layer surfaces a set of APIs used when implementing code analysis and refactoring tools that function within a host environment like the Visual Studio IDE. Examples include the Find All References, Formatting, and Code Generation APIs.

This layer has no dependencies on Visual Studio components.

Work with syntax

12/28/2021 • 8 minutes to read • [Edit Online](#)

The *syntax tree* is a fundamental immutable data structure exposed by the compiler APIs. These trees represent the lexical and syntactic structure of source code. They serve two important purposes:

- To allow tools - such as an IDE, add-ins, code analysis tools, and refactorings - to see and process the syntactic structure of source code in a user's project.
- To enable tools - such as refactorings and an IDE - to create, modify, and rearrange source code in a natural manner without having to use direct text edits. By creating and manipulating trees, tools can easily create and rearrange source code.

Syntax trees

Syntax trees are the primary structure used for compilation, code analysis, binding, refactoring, IDE features, and code generation. No part of the source code is understood without it first being identified and categorized into one of many well-known structural language elements.

Syntax trees have three key attributes:

- They hold all the source information in full fidelity. Full fidelity means that the syntax tree contains every piece of information found in the source text, every grammatical construct, every lexical token, and everything else in between, including white space, comments, and preprocessor directives. For example, each literal mentioned in the source is represented exactly as it was typed. Syntax trees also capture errors in source code when the program is incomplete or malformed by representing skipped or missing tokens.
- They can produce the exact text that they were parsed from. From any syntax node, it's possible to get the text representation of the subtree rooted at that node. This ability means that syntax trees can be used as a way to construct and edit source text. By creating a tree you have, by implication, created the equivalent text, and by making a new tree out of changes to an existing tree, you have effectively edited the text.
- They are immutable and thread-safe. After a tree is obtained, it's a snapshot of the current state of the code and never changes. This allows multiple users to interact with the same syntax tree at the same time in different threads without locking or duplication. Because the trees are immutable and no modifications can be made directly to a tree, factory methods help create and modify syntax trees by creating additional snapshots of the tree. The trees are efficient in the way they reuse underlying nodes, so a new version can be rebuilt fast and with little extra memory.

A syntax tree is literally a tree data structure, where non-terminal structural elements parent other elements. Each syntax tree is made up of nodes, tokens, and trivia.

Syntax nodes

Syntax nodes are one of the primary elements of syntax trees. These nodes represent syntactic constructs such as declarations, statements, clauses, and expressions. Each category of syntax nodes is represented by a separate class derived from [Microsoft.CodeAnalysis.SyntaxNode](#). The set of node classes is not extensible.

All syntax nodes are non-terminal nodes in the syntax tree, which means they always have other nodes and tokens as children. As a child of another node, each node has a parent node that can be accessed through the [SyntaxNode.Parent](#) property. Because nodes and trees are immutable, the parent of a node never changes. The root of the tree has a null parent.

Each node has a [SyntaxNode.ChildNodes\(\)](#) method, which returns a list of child nodes in sequential order based

on their position in the source text. This list does not contain tokens. Each node also has methods to examine Descendants, such as [DescendantNodes](#), [DescendantTokens](#), or [DescendantTrivia](#) - that represent a list of all the nodes, tokens, or trivia that exist in the subtree rooted by that node.

In addition, each syntax node subclass exposes all the same children through strongly typed properties. For example, a [BinaryExpressionSyntax](#) node class has three additional properties specific to binary operators: [Left](#), [OperatorToken](#), and [Right](#). The type of [Left](#) and [Right](#) is [ExpressionSyntax](#), and the type of [OperatorToken](#) is [SyntaxToken](#).

Some syntax nodes have optional children. For example, an [IfStatementSyntax](#) has an optional [ElseClauseSyntax](#). If the child is not present, the property returns null.

Syntax tokens

Syntax tokens are the terminals of the language grammar, representing the smallest syntactic fragments of the code. They are never parents of other nodes or tokens. Syntax tokens consist of keywords, identifiers, literals, and punctuation.

For efficiency purposes, the [SyntaxToken](#) type is a CLR value type. Therefore, unlike syntax nodes, there is only one structure for all kinds of tokens with a mix of properties that have meaning depending on the kind of token that is being represented.

For example, an integer literal token represents a numeric value. In addition to the raw source text the token spans, the literal token has a [Value](#) property that tells you the exact decoded integer value. This property is typed as [Object](#) because it may be one of many primitive types.

The [ValueText](#) property tells you the same information as the [Value](#) property; however this property is always typed as [String](#). An identifier in C# source text may include Unicode escape characters, yet the syntax of the escape sequence itself is not considered part of the identifier name. So although the raw text spanned by the token does include the escape sequence, the [ValueText](#) property does not. Instead, it includes the Unicode characters identified by the escape. For example, if the source text contains an identifier written as `\u03C0`, then the [ValueText](#) property for this token will return `π`.

Syntax trivia

Syntax trivia represent the parts of the source text that are largely insignificant for normal understanding of the code, such as white space, comments, and preprocessor directives. Like syntax tokens, trivia are value types. The single [Microsoft.CodeAnalysis.SyntaxTrivia](#) type is used to describe all kinds of trivia.

Because trivia are not part of the normal language syntax and can appear anywhere between any two tokens, they are not included in the syntax tree as a child of a node. Yet, because they are important when implementing a feature like refactoring and to maintain full fidelity with the source text, they do exist as part of the syntax tree.

You can access trivia by inspecting a token's [SyntaxToken.LeadingTrivia](#) or [SyntaxToken.TrailingTrivia](#) collections. When source text is parsed, sequences of trivia are associated with tokens. In general, a token owns any trivia after it on the same line up to the next token. Any trivia after that line is associated with the following token. The first token in the source file gets all the initial trivia, and the last sequence of trivia in the file is tacked onto the end-of-file token, which otherwise has zero width.

Unlike syntax nodes and tokens, syntax trivia do not have parents. Yet, because they are part of the tree and each is associated with a single token, you may access the token it is associated with using the [SyntaxTrivia.Token](#) property.

Spans

Each node, token, or trivia knows its position within the source text and the number of characters it consists of. A

text position is represented as a 32-bit integer, which is a zero-based `char` index. A `TextSpan` object is the beginning position and a count of characters, both represented as integers. If `TextSpan` has a zero length, it refers to a location between two characters.

Each node has two `TextSpan` properties: `Span` and `FullSpan`.

The `Span` property is the text span from the start of the first token in the node's subtree to the end of the last token. This span does not include any leading or trailing trivia.

The `FullSpan` property is the text span that includes the node's normal span, plus the span of any leading or trailing trivia.

For example:

```
    if (x > 3)
    {
||      // this is bad
        |throw new Exception("Not right.");| // better exception?||
    }
```

The statement node inside the block has a span indicated by the single vertical bars (`|`). It includes the characters `throw new Exception("Not right.");`. The full span is indicated by the double vertical bars (`||`). It includes the same characters as the span and the characters associated with the leading and trailing trivia.

Kinds

Each node, token, or trivia has a `SyntaxNode.RawKind` property, of type `System.Int32`, that identifies the exact syntax element represented. This value can be cast to a language-specific enumeration. Each language, C# or Visual Basic, has a single `SyntaxKind` enumeration (`Microsoft.CodeAnalysis.CSharp.SyntaxKind` and `Microsoft.CodeAnalysis.VisualBasic.SyntaxKind`, respectively) that lists all the possible nodes, tokens, and trivia elements in the grammar. This conversion can be done automatically by accessing the `CSharpExtensions.Kind` or `VisualBasicExtensions.Kind` extension methods.

The `RawKind` property allows for easy disambiguation of syntax node types that share the same node class. For tokens and trivia, this property is the only way to distinguish one type of element from another.

For example, a single `BinaryExpressionSyntax` class has `Left`, `OperatorToken`, and `Right` as children. The `Kind` property distinguishes whether it is an `AddExpression`, `SubtractExpression`, or `MultiplyExpression` kind of syntax node.

TIP

It's recommended to check kinds using `IsKind` (for C#) or `IsKind` (for VB) extension methods.

Errors

Even when the source text contains syntax errors, a full syntax tree that is round-trippable to the source is exposed. When the parser encounters code that does not conform to the defined syntax of the language, it uses one of two techniques to create a syntax tree:

- If the parser expects a particular kind of token but does not find it, it may insert a missing token into the syntax tree in the location that the token was expected. A missing token represents the actual token that was expected, but it has an empty span, and its `SyntaxNode.IsMissing` property returns `true`.
- The parser may skip tokens until it finds one where it can continue parsing. In this case, the skipped tokens are attached as a trivia node with the kind `SkippedTokensTrivia`.

Work with semantics

12/28/2021 • 3 minutes to read • [Edit Online](#)

[Syntax trees](#) represent the lexical and syntactic structure of source code. Although this information alone is enough to describe all the declarations and logic in the source, it is not enough information to identify what is being referenced. A name may represent:

- a type
- a field
- a method
- a local variable

Although each of these is uniquely different, determining which one an identifier actually refers to often requires a deep understanding of the language rules.

There are program elements represented in source code, and programs can also refer to previously compiled libraries, packaged in assembly files. Although no source code, and therefore no syntax nodes or trees, are available for assemblies, programs can still refer to elements inside them.

For those tasks, you need the **Semantic model**.

In addition to a syntactic model of the source code, a semantic model encapsulates the language rules, giving you an easy way to correctly match identifiers with the correct program element being referenced.

Compilation

A compilation is a representation of everything needed to compile a C# or Visual Basic program, which includes all the assembly references, compiler options, and source files.

Because all this information is in one place, the elements contained in the source code can be described in more detail. The compilation represents each declared type, member, or variable as a symbol. The compilation contains a variety of methods that help you find and relate the symbols that have either been declared in the source code or imported as metadata from an assembly.

Similar to syntax trees, compilations are immutable. After you create a compilation, it cannot be changed by you or anyone else you might be sharing it with. However, you can create a new compilation from an existing compilation, specifying a change as you do so. For example, you might create a compilation that is the same in every way as an existing compilation, except it may include an additional source file or assembly reference.

Symbols

A symbol represents a distinct element declared by the source code or imported from an assembly as metadata. Every namespace, type, method, property, field, event, parameter, or local variable is represented by a symbol.

A variety of methods and properties on the [Compilation](#) type help you find symbols. For example, you can find a symbol for a declared type by its common metadata name. You can also access the entire symbol table as a tree of symbols rooted by the global namespace.

Symbols also contain additional information that the compiler determines from the source or metadata, such as other referenced symbols. Each kind of symbol is represented by a separate interface derived from [ISymbol](#), each with its own methods and properties detailing the information the compiler has gathered. Many of these properties directly reference other symbols. For example, the [IMethodSymbol.ReturnType](#) property tells you the

actual type symbol that the method returns.

Symbols present a common representation of namespaces, types, and members, between source code and metadata. For example, a method that was declared in source code and a method that was imported from metadata are both represented by an [IMethodSymbol](#) with the same properties.

Symbols are similar in concept to the CLR type system as represented by the [System.Reflection](#) API, yet they are richer in that they model more than just types. Namespaces, local variables, and labels are all symbols. In addition, symbols are a representation of language concepts, not CLR concepts. There is a lot of overlap, but there are many meaningful distinctions as well. For instance, an iterator method in C# or Visual Basic is a single symbol. However, when the iterator method is translated to CLR metadata, it is a type and multiple methods.

Semantic model

A semantic model represents all the semantic information for a single source file. You can use it to discover the following:

- The symbols referenced at a specific location in source.
- The resultant type of any expression.
- All diagnostics, which are errors and warnings.
- How variables flow in and out of regions of source.
- The answers to more speculative questions.

Work with a workspace

12/28/2021 • 2 minutes to read • [Edit Online](#)

The **Workspaces** layer is the starting point for doing code analysis and refactoring over entire solutions. Within this layer, the Workspace API assists you in organizing all the information about the projects in a solution into a single object model, offering you direct access to compiler layer object models like source text, syntax trees, semantic models, and compilations without needing to parse files, configure options, or manage inter-project dependencies.

Host environments, like an IDE, provide a workspace for you corresponding to the open solution. It is also possible to use this model outside of an IDE by simply loading a solution file.

Workspace

A workspace is an active representation of your solution as a collection of projects, each with a collection of documents. A workspace is typically tied to a host environment that is constantly changing as a user types or manipulates properties.

The **Workspace** provides access to the current model of the solution. When a change in the host environment occurs, the workspace fires corresponding events, and the **Workspace.CurrentSolution** property is updated. For example, when the user types in a text editor corresponding to one of the source documents, the workspace uses an event to signal that the overall model of the solution has changed and which document was modified. You can then react to those changes by analyzing the new model for correctness, highlighting areas of significance, or making a suggestion for a code change.

You can also create stand-alone workspaces that are disconnected from the host environment or used in an application that has no host environment.

Solutions, projects, and documents

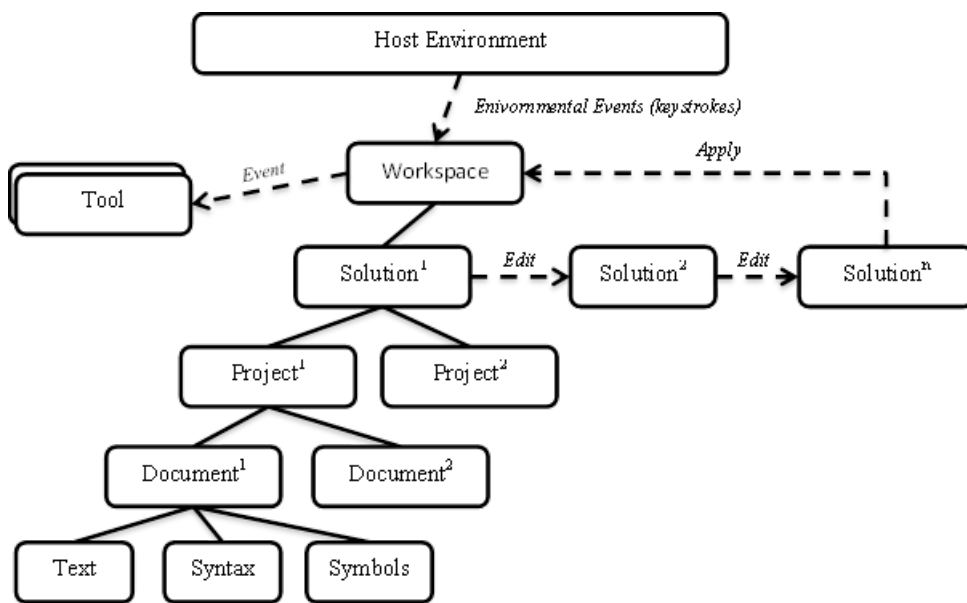
Although a workspace may change every time a key is pressed, you can work with the model of the solution in isolation.

A solution is an immutable model of the projects and documents. This means that the model can be shared without locking or duplication. After you obtain a solution instance from the **Workspace.CurrentSolution** property, that instance will never change. However, like with syntax trees and compilations, you can modify solutions by constructing new instances based on existing solutions and specific changes. To get the workspace to reflect your changes, you must explicitly apply the changed solution back to the workspace.

A project is a part of the overall immutable solution model. It represents all the source code documents, parse and compilation options, and both assembly and project-to-project references. From a project, you can access the corresponding compilation without needing to determine project dependencies or parse any source files.

A document is also a part of the overall immutable solution model. A document represents a single source file from which you can access the text of the file, the syntax tree, and the semantic model.

The following diagram is a representation of how the Workspace relates to the host environment, tools, and how edits are made.



Summary

Roslyn exposes a set of compiler APIs and Workspaces APIs that provides rich information about your source code and that has full fidelity with the C# and Visual Basic languages. The .NET Compiler Platform SDK dramatically lowers the barrier to entry for creating code-focused tools and applications. It creates many opportunities for innovation in areas such as meta-programming, code generation and transformation, interactive use of the C# and Visual Basic languages, and embedding of C# and Visual Basic in domain-specific languages.

Explore code with the Roslyn syntax visualizer in Visual Studio

12/28/2021 • 9 minutes to read • [Edit Online](#)

This article provides an overview of the Syntax Visualizer tool that ships as part of the .NET Compiler Platform ("Roslyn") SDK. The Syntax Visualizer is a tool window that helps you inspect and explore syntax trees. It's an essential tool to understand the models for code you want to analyze. It's also a debugging aid when you develop your own applications using the .NET Compiler Platform ("Roslyn") SDK. Open this tool as you create your first analyzers. The visualizer helps you understand the models used by the APIs. You can also use tools like [SharpLab](#) or [LINQPad](#) to inspect code and understand syntax trees.

Installation instructions - Visual Studio Installer

There are two different ways to find the .NET Compiler Platform SDK in the Visual Studio Installer:

Install using the Visual Studio Installer - Workloads view

The .NET Compiler Platform SDK is not automatically selected as part of the Visual Studio extension development workload. You must select it as an optional component.

1. Run **Visual Studio Installer**
2. Select **Modify**
3. Check the **Visual Studio extension development** workload.
4. Open the **Visual Studio extension development** node in the summary tree.
5. Check the box for **.NET Compiler Platform SDK**. You'll find it last under the optional components.

Optionally, you'll also want the **DGML editor** to display graphs in the visualizer:

1. Open the **Individual components** node in the summary tree.
2. Check the box for **DGML editor**

Install using the Visual Studio Installer - Individual components tab

1. Run **Visual Studio Installer**
2. Select **Modify**
3. Select the **Individual components** tab
4. Check the box for **.NET Compiler Platform SDK**. You'll find it at the top under the **Compilers, build tools, and runtimes** section.

Optionally, you'll also want the **DGML editor** to display graphs in the visualizer:

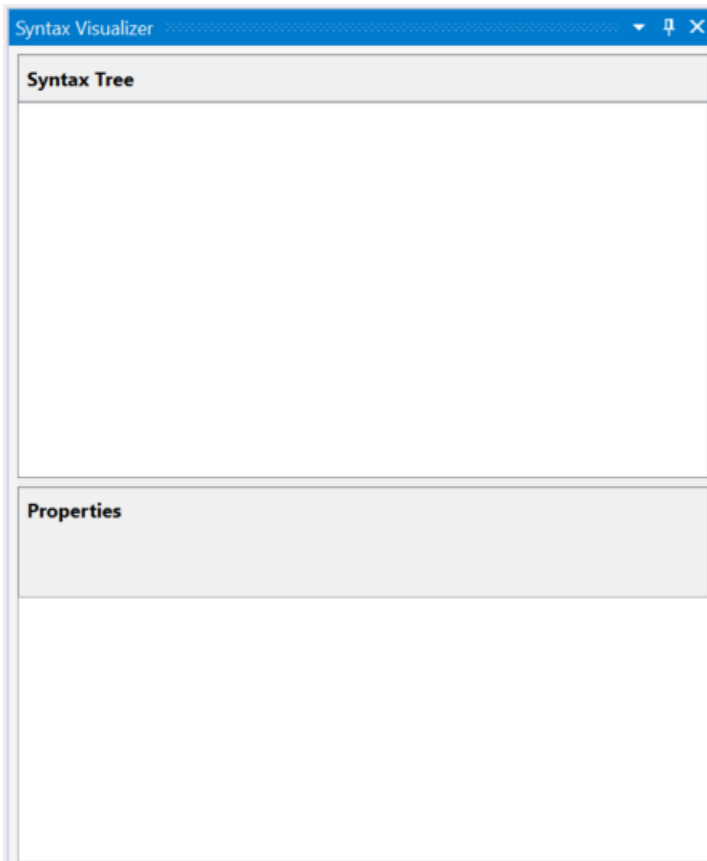
1. Check the box for **DGML editor**. You'll find it under the **Code tools** section.

Familiarize yourself with the concepts used in the .NET Compiler Platform SDK by reading the [overview](#) article. It provides an introduction to syntax trees, nodes, tokens, and trivia.

Syntax Visualizer

The **Syntax Visualizer** enables inspection of the syntax tree for the C# or Visual Basic code file in the current active editor window inside the Visual Studio IDE. The visualizer can be launched by clicking on **View > Other Windows > Syntax Visualizer**. You can also use the **Quick Launch** toolbar in the upper right corner. Type "syntax", and the command to open the **Syntax Visualizer** should appear.

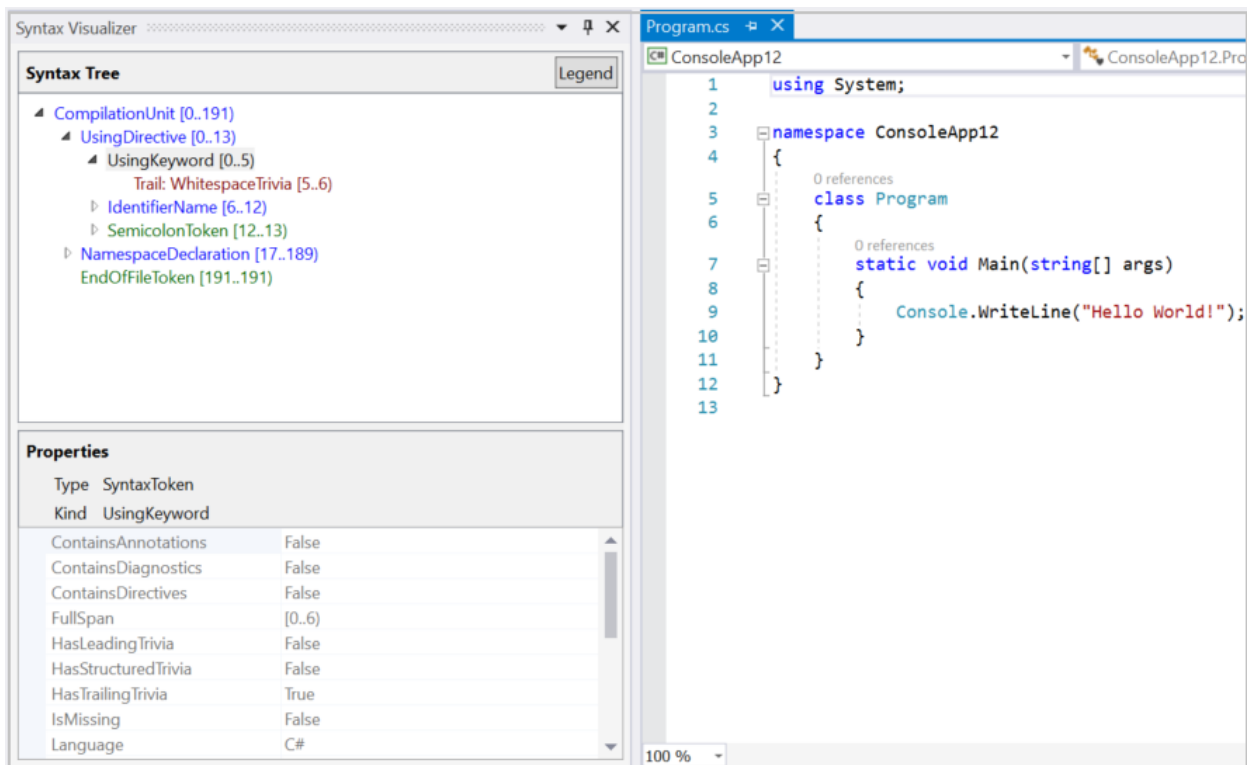
This command opens the Syntax Visualizer as a floating tool window. If you don't have a code editor window open, the display is blank, as shown in the following figure.



Dock this tool window at a convenient location inside Visual Studio, such as the left side. The Visualizer shows information about the current code file.

Create a new project using the **File > New Project** command. You can create either a Visual Basic or C# project. When Visual Studio opens the main code file for this project, the visualizer displays the syntax tree for it. You can open any existing C# / Visual Basic file in this Visual Studio instance, and the visualizer displays that file's syntax tree. If you have multiple code files open inside Visual Studio, the visualizer displays the syntax tree for the currently active code file, (the code file that has keyboard focus.)

- [C#](#)
- [Visual Basic](#)



As shown in the preceding images, the visualizer tool window displays the syntax tree at the top and a property grid at the bottom. The property grid displays the properties of the item that is currently selected in the tree, including the .NET *Type* and the *Kind* (SyntaxKind) of the item.

Syntax trees comprise three types of items – *nodes*, *tokens*, and *trivia*. You can read more about these types in the [Work with syntax](#) article. Items of each type are represented using a different color. Click on the 'Legend' button for an overview of the colors used.

Each item in the tree also displays its own **span**. The **span** is the indices (the starting and ending position) of that node in the text file. In the preceding C# example, the selected "UsingKeyword [0..5)" token has a **Span** that is five characters wide, [0..5). The "[..)" notation means that the starting index is part of the span, but the ending index is not.

There are two ways to navigate the tree:

- Expand or click on items in the tree. The visualizer automatically selects the text corresponding to this item's span in the code editor.
- Click or select text in the code editor. In the preceding Visual Basic example, if you select the line containing "Module Module1" in the code editor, the visualizer automatically navigates to the corresponding ModuleStatement node in the tree.

The visualizer highlights the item in the tree whose span best matches the span of the text selected in the editor.

The visualizer refreshes the tree to match modifications in the active code file. Add a call to `Console.WriteLine()` inside `Main()`. As you type, the visualizer refreshes the tree.

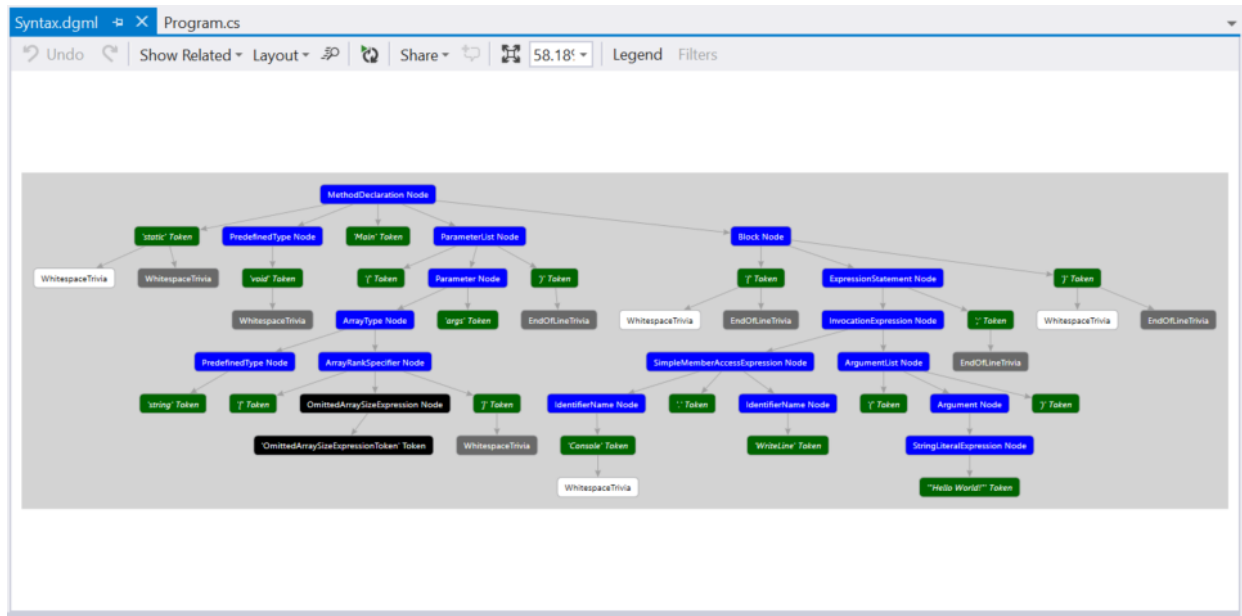
Pause typing once you have typed `Console.`. The tree has some items colored in pink. At this point, there are errors (also referred to as 'Diagnostics') in the typed code. These errors are attached to nodes, tokens, and trivia in the syntax tree. The visualizer shows you which items have errors attached to them highlighting the background in pink. You can inspect the errors on any item colored pink by hovering over the item. The visualizer only displays syntactic errors (those errors related to the syntax of the typed code); it doesn't display any semantic errors.

Syntax Graphs

Right click on any item in the tree and click on **View Directed Syntax Graph**.

- C#
- Visual Basic

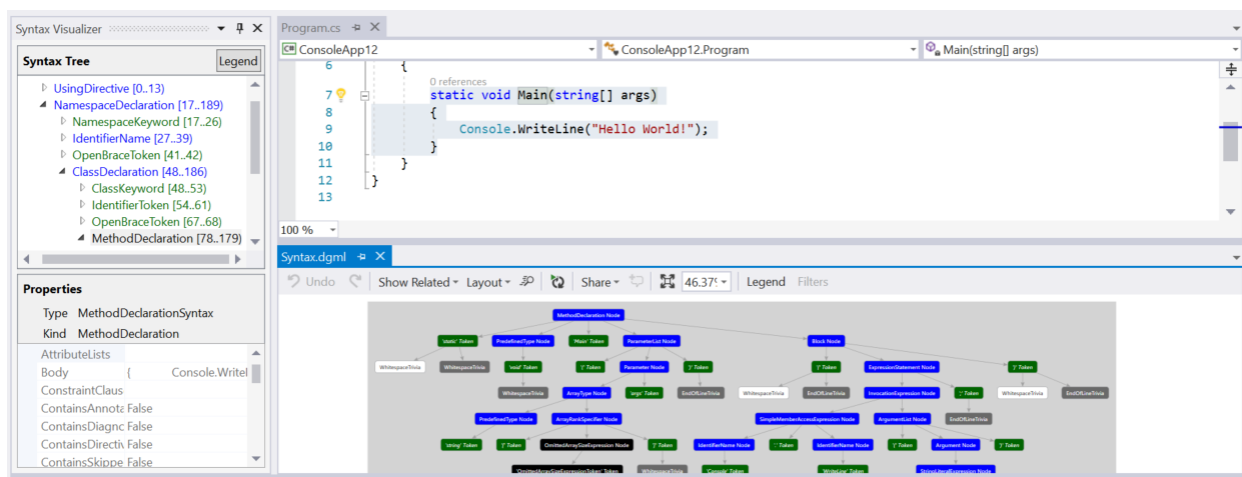
The visualizer displays a graphical representation of the subtree rooted at the selected item. Try these steps for the **MethodDeclaration** node corresponding to the `Main()` method in the C# example. The visualizer displays a syntax graph that looks as follows:



The syntax graph viewer has an option to display a legend for its coloring scheme. You can also hover over individual items in the syntax graph with the mouse to view the properties corresponding to that item.

You can view syntax graphs for different items in the tree repeatedly and the graphs will always be displayed in the same window inside Visual Studio. You can dock this window at a convenient location inside Visual Studio so that you don't have to switch between tabs to view a new syntax graph. The bottom, below code editor windows, is often convenient.

Here is the docking layout to use with the visualizer tool window and the syntax graph window:



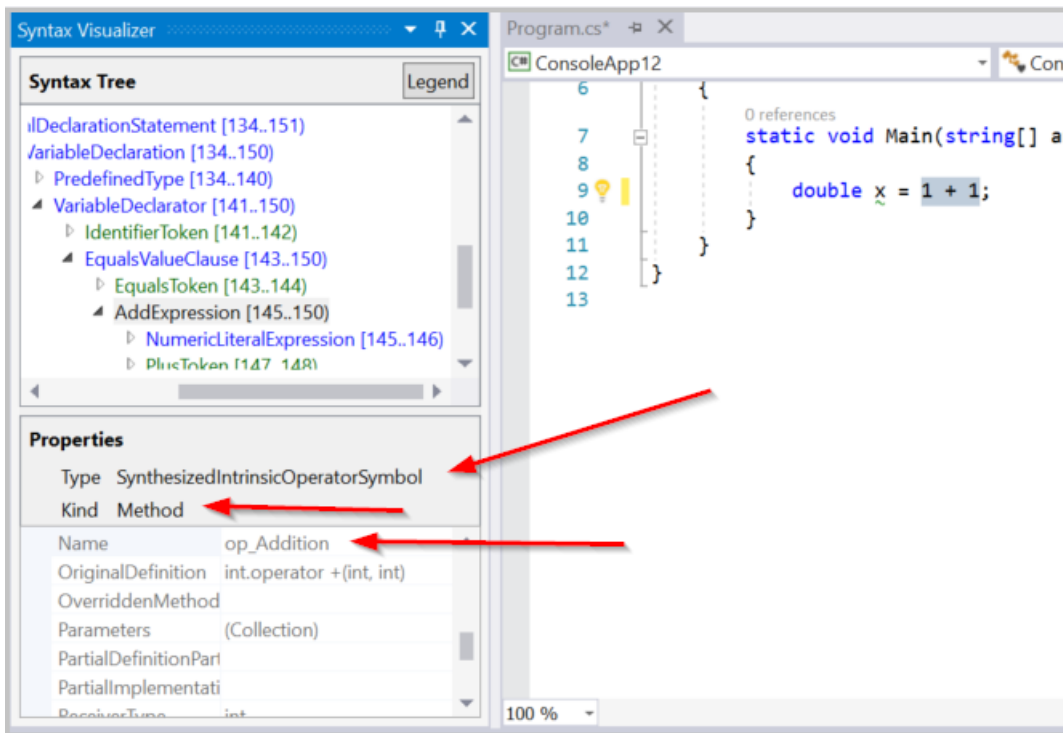
Another option is to put the syntax graph window on a second monitor, in a dual monitor setup.

Inspecting semantics

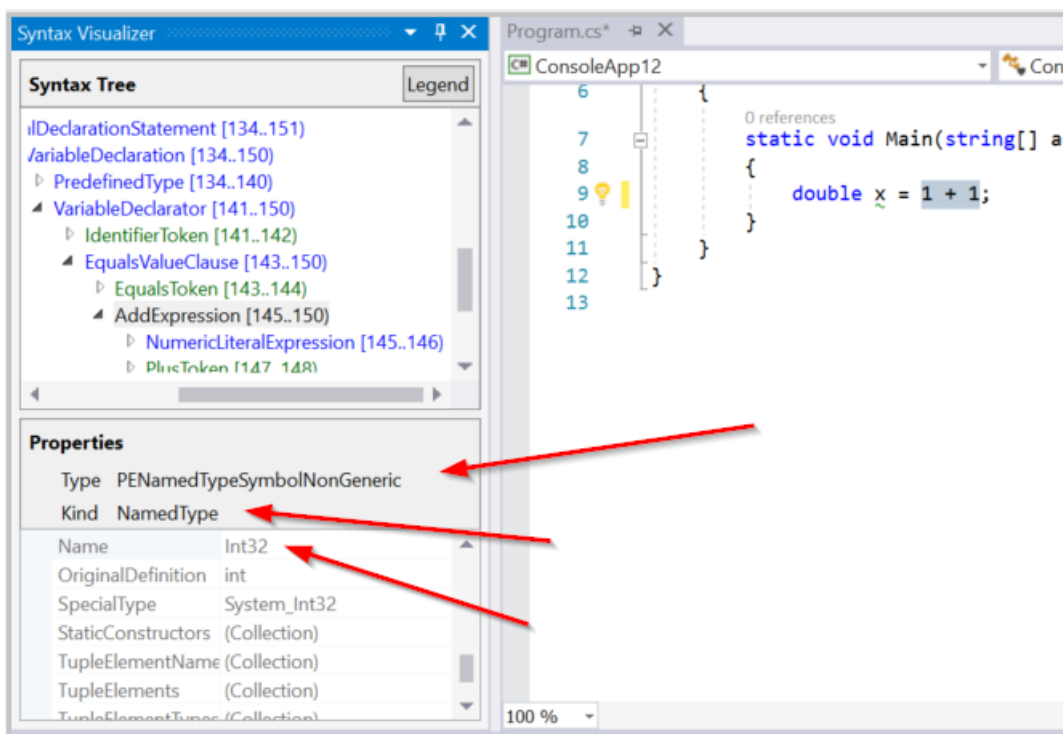
The Syntax Visualizer enables rudimentary inspection of symbols and semantic information. Type `double x = 1 + 1;` inside `Main()` in the C# example. Then, select the expression `1 + 1` in the code editor

window. The visualizer highlights the **AddExpression** node in the visualizer. Right click on this **AddExpression** and click on **View Symbol (if any)**. Notice that most of the menu items have the "if any" qualifier. The Syntax Visualizer inspects properties of a Node, including properties that may not be present for all nodes.

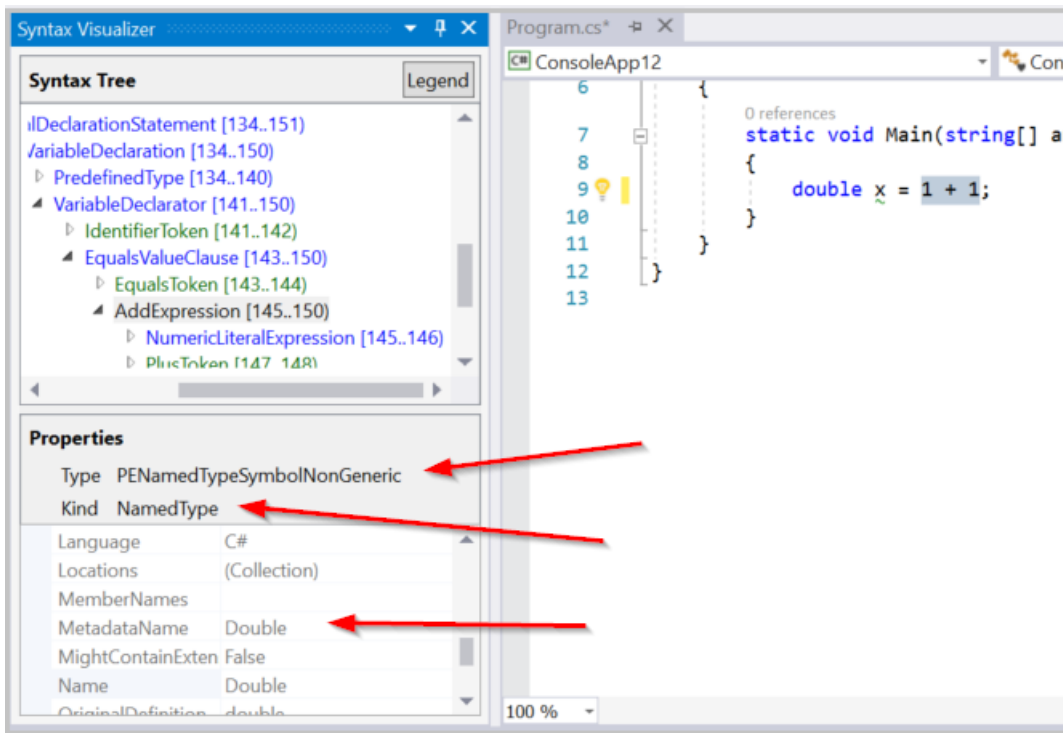
The property grid in the visualizer updates as shown in the following figure: The symbol for the expression is a **SynthesizedIntrinsicOperatorSymbol** with **Kind = Method**.



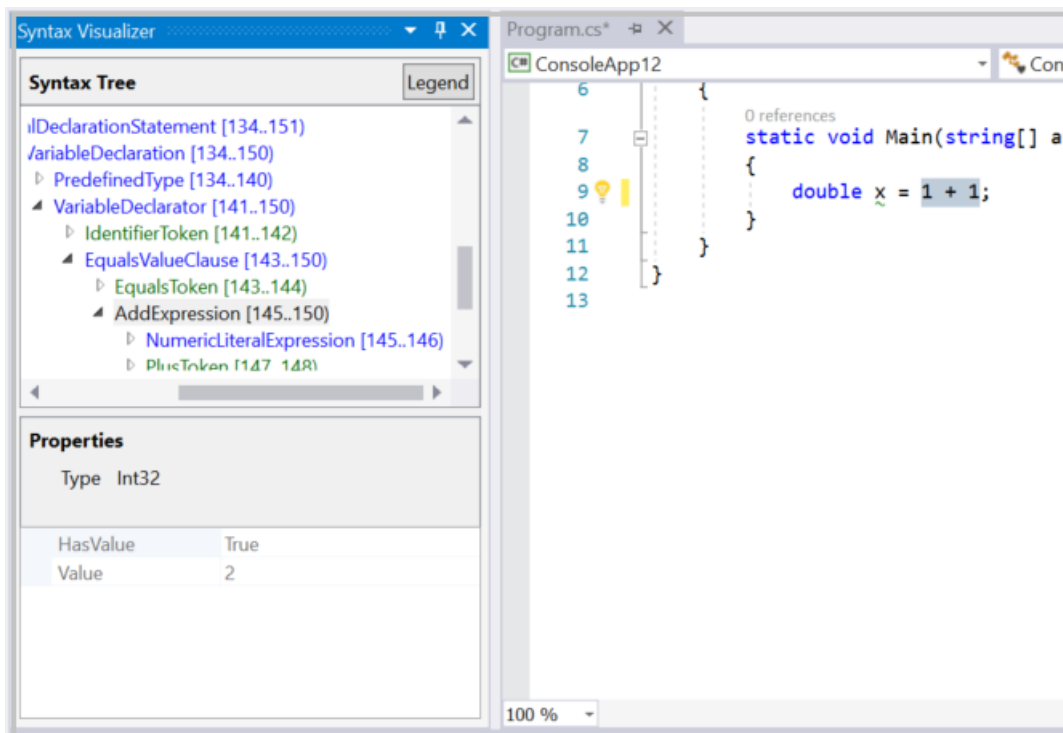
Try **View TypeSymbol (if any)** for the same **AddExpression** node. The property grid in the visualizer updates as shown in the following figure, indicating that the type of the selected expression is **Int32**.



Try **View Converted TypeSymbol (if any)** for the same **AddExpression** node. The property grid updates indicating that although the type of the expression is **Int32**, the converted type of the expression is **Double** as shown in the following figure. This node includes converted type symbol information because the **Int32** expression occurs in a context where it must be converted to a **Double**. This conversion satisfies the **Double** type specified for the variable **x** on the left-hand side of the assignment operator.



Finally, try **View Constant Value (if any)** for the same `AddExpression` node. The property grid shows that the value of the expression is a compile time constant with value `2`.



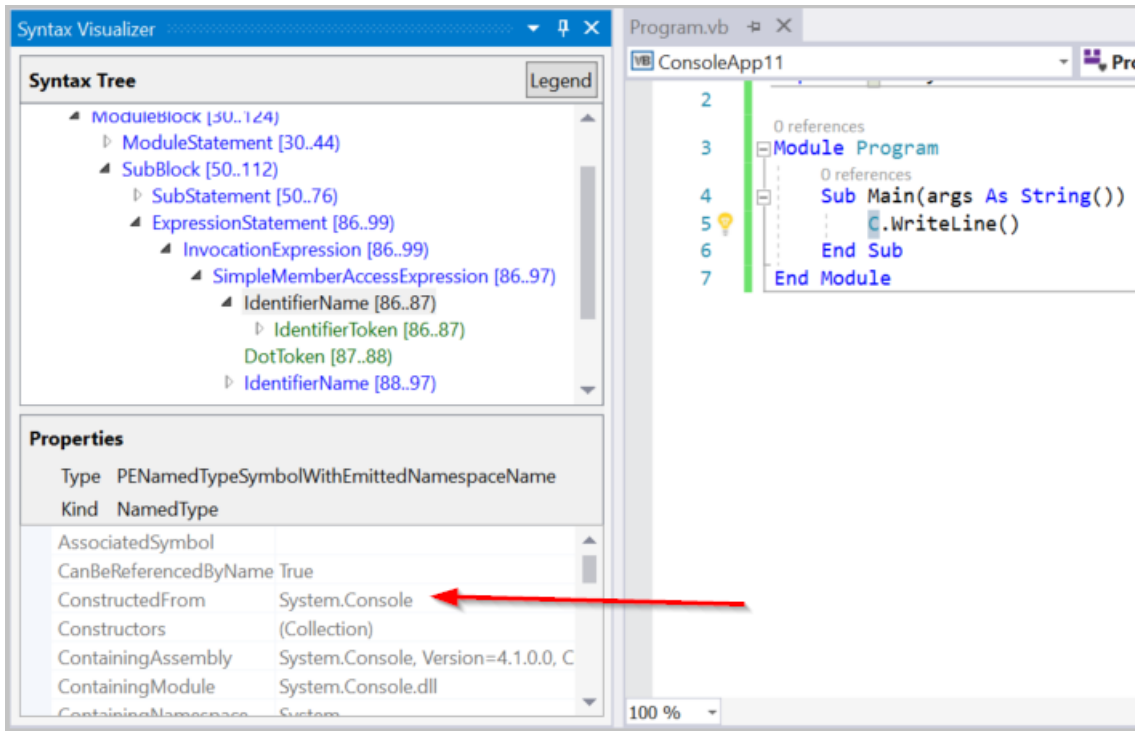
The preceding example can also be replicated in Visual Basic. Type `Dim x As Double = 1 + 1` in a Visual Basic file. Select the expression `1 + 1` in the code editor window. The visualizer highlights the corresponding `AddExpression` node in the visualizer. Repeat the preceding steps for this `AddExpression` and you should see identical results.

Examine more code in Visual Basic. Update your main Visual Basic file with the following code:

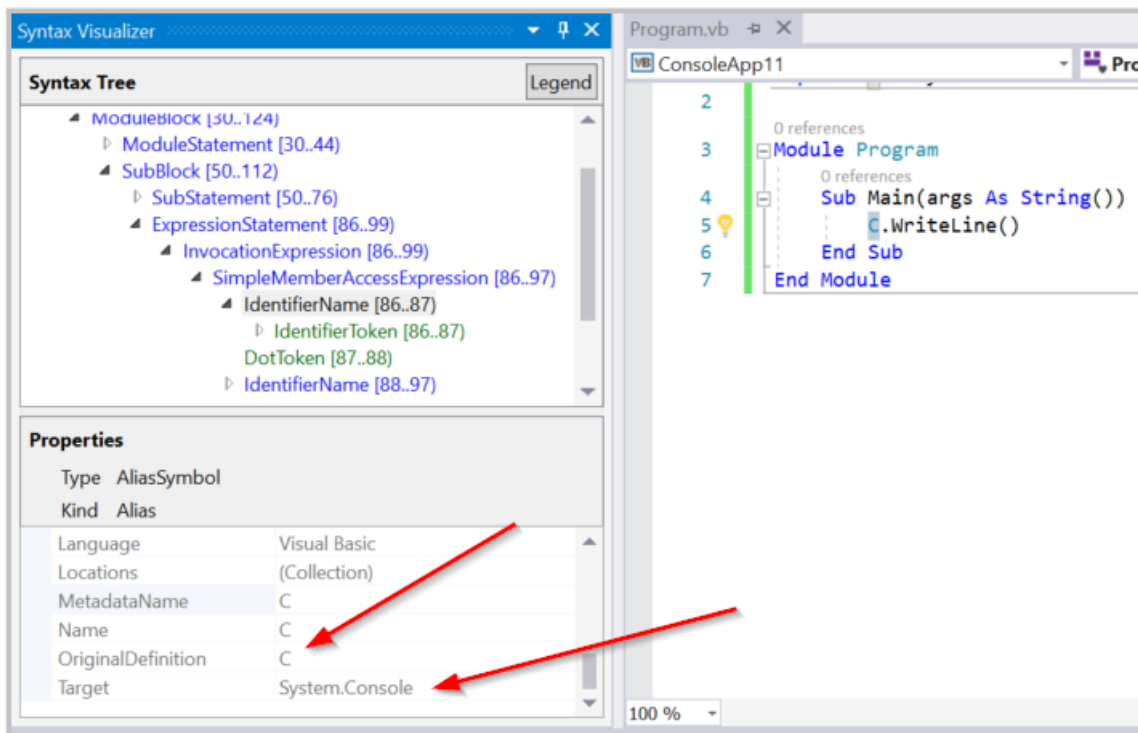

```
Imports C = System.Console

Module Program
    Sub Main(args As String())
        C.WriteLine()
    End Sub
End Module
```

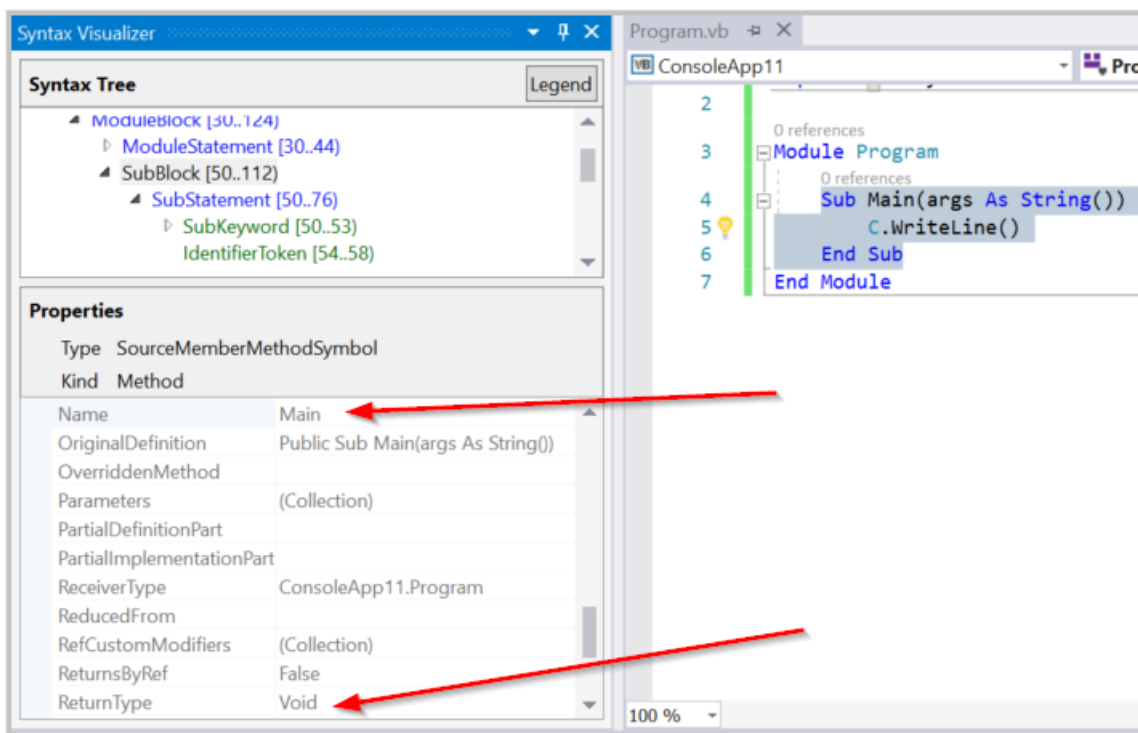
This code introduces an alias named `C` that maps to the type `System.Console` at the top of the file and uses this alias inside `Main()`. Select the use of this alias, the `C` in `C.WriteLine()`, inside the `Main()` method. The visualizer selects the corresponding **IdentifierName** node in the visualizer. Right-click this node and click on **View Symbol (if any)**. The property grid indicates that this identifier is bound to the type `System.Console` as shown in the following figure:



Try **View AliasSymbol (if any)** for the same **IdentifierName** node. The property grid indicates the identifier is an alias with name `C` that is bound to the `System.Console` target. In other words, the property grid provides information regarding the **AliasSymbol** corresponding to the identifier `C`.



Inspect the symbol corresponding to any declared type, method, property. Select the corresponding node in the visualizer and click on **View Symbol (if any)**. Select the method `Sub Main()`, including the body of the method. Click on **View Symbol (if any)** for the corresponding **SubBlock** node in the visualizer. The property grid shows the **MethodSymbol** for this **SubBlock** has name `Main` with return type `Void`.



The above Visual Basic examples can be easily replicated in C#. Type `using C = System.Console;` in place of `Imports C = System.Console` for the alias. The preceding steps in C# yield identical results in the visualizer window.

Semantic inspection operations are only available on nodes. They are not available on tokens or trivia. Not all nodes have interesting semantic information to inspect. When a node doesn't have interesting semantic information, clicking on **View * Symbol (if any)** shows a blank property grid.

You can read more about APIs for performing semantic analysis in the [Work with semantics](#) overview document.

Closing the syntax visualizer

You can close the visualizer window when you are not using it to examine source code. The syntax visualizer updates its display as you navigate through code, editing and changing the source. It can get distracting when you are not using it.

Source Generators

12/28/2021 • 7 minutes to read • [Edit Online](#)

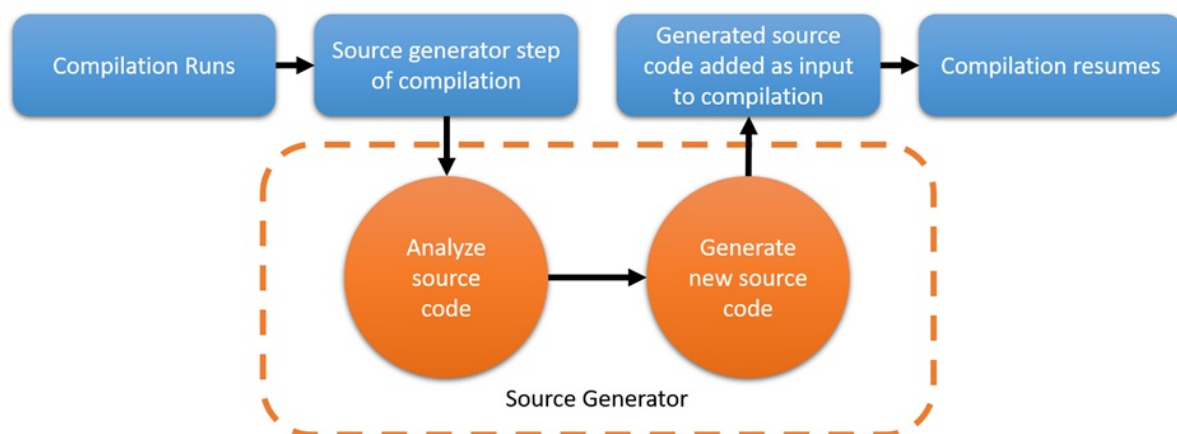
This article provides an overview of Source Generators that ships as part of the .NET Compiler Platform ("Roslyn") SDK. Source Generators are a C# compiler feature that lets C# developers inspect user code as it is being compiled and generate new C# source files on the fly that are added to the user's compilation. In this way, you can have code that runs during compilation and inspects your program to produce additional source files that are compiled together with the rest of your code.

A Source Generator is a new kind of component that C# developers can write that lets you do two major things:

1. Retrieve a *compilation object* that represents all user code that is being compiled. This object can be inspected, and you can write code that works with the syntax and semantic models for the code being compiled, just like with analyzers today.
2. Generate C# source files that can be added to a compilation object during the course of compilation. In other words, you can provide additional source code as input to a compilation while the code is being compiled.

When combined, these two things are what make Source Generators so useful. You can inspect user code with all of the rich metadata that the compiler builds up during compilation, then emit C# code back into the same compilation that is based on the data you've analyzed. If you're familiar with Roslyn Analyzers, you can think of Source Generators as analyzers that can emit C# source code.

Source generators run as a phase of compilation visualized below:



A Source Generator is a .NET Standard 2.0 assembly that is loaded by the compiler along with any analyzers. It is usable in environments where .NET Standard components can be loaded and run.

IMPORTANT

Currently only .NET Standard 2.0 assemblies can be used as Source Generators.

Common scenarios

There are three general approaches to inspecting user code and generating information or code based on that analysis used by technologies today:

- Runtime reflection.
- Juggling MSBuild tasks.
- Intermediate Language (IL) weaving (not discussed in this article).

Source Generators can be an improvement over each approach.

Runtime reflection

Runtime reflection is a powerful technology that was added to .NET a long time ago. There are countless scenarios for using it. A very common scenario is to perform some analysis of user code when an app starts up and use that data to generate things.

For example, ASP.NET Core uses reflection when your web service first runs to discover constructs you've defined so that it can "wire up" things like controllers and razor pages. Although this enables you to write straightforward code with powerful abstractions, it comes with a performance penalty at run time: when your web service or app first starts up, it cannot accept any requests until all the runtime reflection code that discovers information about your code is finished running. Although this performance penalty is not enormous, it is somewhat of a fixed cost that you cannot improve yourself in your own app.

With a Source Generator, the controller discovery phase of startup could instead happen at compile time by analyzing your source code and emitting the code it needs to "wire up" your app. This could result in some faster startup times, since an action happening at run time today could get pushed into compile time.

Juggling MSBuild tasks

Source Generators can improve performance in ways that aren't limited to reflection at run time to discover types as well. Some scenarios involve calling the MSBuild C# task (called CSC) multiple times so they can inspect data from a compilation. As you might imagine, calling the compiler more than once affects the total time it takes to build your app. We're investigating how Source Generators can be used to obviate the need for juggling MSBuild tasks like this, since Source generators don't just offer some performance benefits, but also allows tools to operate at the right level of abstraction.

Another capability Source Generators can offer is obviating the use of some "stringly-typed" APIs, such as how ASP.NET Core routing between controllers and razor pages work. With a Source Generator, routing can be strongly typed with the necessary strings being generated as a compile-time detail. This would reduce the amount of times a mistyped string literal leads to a request not hitting the correct controller.

Get started with source generators

In this guide, you'll explore the creation of a source generator using the [ISourceGenerator](#) API.

1. Create a .NET console application. This example uses .NET 6.
2. Replace the `Program` class with the following:

```
namespace ConsoleApp;

partial class Program
{
    static void Main(string[] args)
    {
        HelloFrom("Generated Code");
    }

    static partial void HelloFrom(string name);
}
```

NOTE

You can run this sample as-is, but nothing will happen yet.

- Next, we'll create a source generator project that will implement the `partial void HelloFrom` method counterpart.
- Create a .NET standard library project that targets the `netstandard2.0` target framework moniker (TFM):

```
<Project Sdk="Microsoft.NET.Sdk">

  <PropertyGroup>
    <TargetFramework>netstandard2.0</TargetFramework>
  </PropertyGroup>

  <ItemGroup>
    <PackageReference Include="Microsoft.CodeAnalysis.CSharp" Version="4.0.1" PrivateAssets="all" />
    <PackageReference Include="Microsoft.CodeAnalysis.Analyzers" Version="3.3.3" PrivateAssets="all" />
  </ItemGroup>

</Project>
```

TIP

The source generator project needs to target the `netstandard2.0` TFM, otherwise it will not work.

- Create a new C# file named *HelloSourceGenerator.cs* that specifies your own Source Generator like so:

```
using Microsoft.CodeAnalysis;

namespace SourceGenerator
{
    [Generator]
    public class HelloSourceGenerator : ISourceGenerator
    {
        public void Execute(GeneratorExecutionContext context)
        {
            // Code generation goes here
        }

        public void Initialize(GeneratorInitializationContext context)
        {
            // No initialization required for this one
        }
    }
}
```

A source generator needs to both implement the [Microsoft.CodeAnalysis.ISourceGenerator](#) interface, and have the [Microsoft.CodeAnalysis.GeneratorAttribute](#). Not all source generators require initialization, and that is the case with this example implementation — where [ISourceGenerator.Initialize](#) is empty.

- Replace the contents of the [ISourceGenerator.Execute](#) method, with the following implementation:

```

using Microsoft.CodeAnalysis;

namespace SourceGenerator
{
    [Generator]
    public class HelloSourceGenerator : ISourceGenerator
    {
        public void Execute(GeneratorExecutionContext context)
        {
            // Find the main method
            var mainMethod = context.Compilation.GetEntryPoint(context.CancellationToken);

            // Build up the source code
            string source = $"@" // Auto-generated code
using System;

namespace {mainMethod.ContainingNamespace.ToDisplayString()}
{{
    public static partial class {mainMethod.ContainingType.Name}
    {{
        static partial void HelloFrom(string name) =>
            Console.WriteLine($"Generator says: Hi from '{{name}}'");
    }}
}}
";

            var typeName = mainMethod.ContainingType.Name;

            // Add the source code to the compilation
            context.AddSource($"{typeName}.g.cs", source);
        }

        public void Initialize(GeneratorInitializationContext context)
        {
            // No initialization required for this one
        }
    }
}

```

From the `context` object we can access the compilations' entry point, or `Main` method. The `mainMethod` instance is an `IMethodSymbol`, and it represents a method or method-like symbol (including constructor, destructor, operator, or property/event accessor). From this object we can reason about the containing namespace (if one is present) and the type. The `source` in this example is an interpolated string that templates the source code to be generated, where the interpolated wholes are filled with the containing namespace and type information. The `source` is added to the `context` with a hint name.

TIP

The `hintName` parameter from the `GeneratorExecutionContext.AddSource` method can be any unique name. It's common to provide an explicit C# file extension such as `".g.cs"` or `".generated.cs"` for the name. The file name helps identify the file as being source generated.

7. We now have a functioning generator, but need to connect it to our console application. Edit the original console application project and add the following, replacing the project path with the one from the .NET Standard project you created above:

```

<!-- Add this as a new ItemGroup, replacing paths and names appropriately -->
<ItemGroup>
  <ProjectReference Include="..\PathTo\SourceGenerator.csproj"
                    OutputItemType="Analyzer"
                    ReferenceOutputAssembly="false" />
</ItemGroup>

```

This is not a traditional project reference, and has to be manually edited to include the `OutputItemType` and `ReferenceOutputAssembly` attributes. For additional information on the `OutputItemType` and `ReferenceOutputAssembly` attributes of `ProjectReference`, see [Common MSBuild project items: ProjectReference](#).

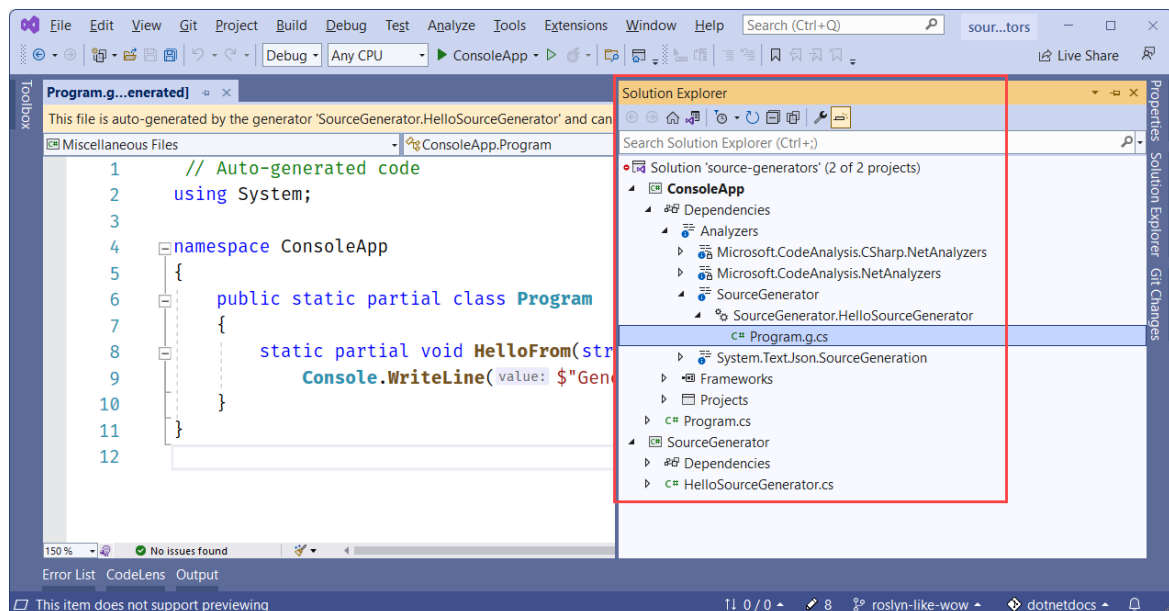
- Now, when you run the console application, you should see that the generated code gets run and prints to the screen. The console application itself doesn't implement the `HelloFrom` method, instead it is source generated during compilation from the Source Generator project. The following is an example output from the application:

```
Generator says: Hi from 'Generated Code'
```

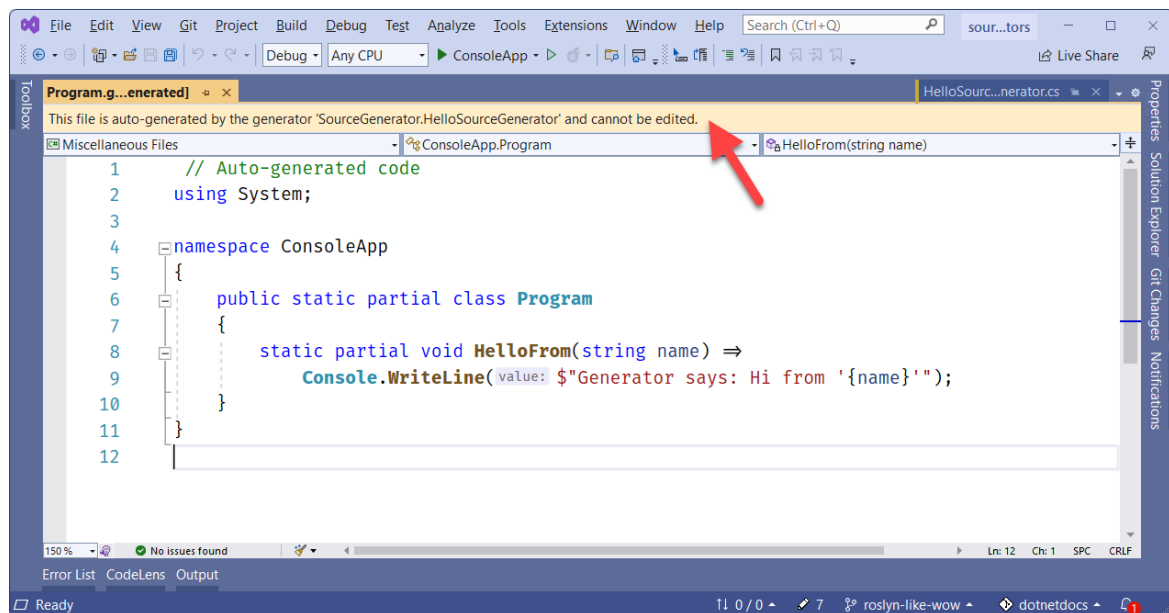
NOTE

You might need to restart Visual Studio to see IntelliSense and get rid of errors as the tooling experience is actively being improved.

- If you're using Visual Studio, you can see the source generated files. From the **Solution Explorer** window expand the **Dependencies > Analyzers > SourceGenerator > SourceGenerator.HelloSourceGenerator**, and double-click the `Program.g.cs` file.



When you open this generated file, Visual Studio will indicate that the file is auto-generated and that it cannot be edited.



Next steps

The [Source Generators Cookbook](#) goes over some of these examples with some recommended approaches to solving them. Additionally, we have a [set of samples available on GitHub](#) that you can try on your own.

You can learn more about Source Generators in these topics:

- [Source Generators design document](#)
- [Source Generators cookbook](#)

Get started with syntax analysis

12/28/2021 • 13 minutes to read • [Edit Online](#)

In this tutorial, you'll explore the **Syntax API**. The Syntax API provides access to the data structures that describe a C# or Visual Basic program. These data structures have enough detail that they can fully represent any program of any size. These structures can describe complete programs that compile and run correctly. They can also describe incomplete programs, as you write them, in the editor.

To enable this rich expression, the data structures and APIs that make up the Syntax API are necessarily complex. Let's start with what the data structure looks like for the typical "Hello World" program:

```
using System;
using System.Collections.Generic;
using System.Linq;

namespace HelloWorld
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Hello World!");
        }
    }
}
```

Look at the text of the previous program. You recognize familiar elements. The entire text represents a single source file, or a **compilation unit**. The first three lines of that source file are **using directives**. The remaining source is contained in a **namespace declaration**. The namespace declaration contains a child **class declaration**. The class declaration contains one **method declaration**.

The Syntax API creates a tree structure with the root representing the compilation unit. Nodes in the tree represent the using directives, namespace declaration and all the other elements of the program. The tree structure continues down to the lowest levels: the string "Hello World!" is a **string literal token** that is a descendent of an **argument**. The Syntax API provides access to the structure of the program. You can query for specific code practices, walk the entire tree to understand the code, and create new trees by modifying the existing tree.

That brief description provides an overview of the kind of information accessible using the Syntax API. The Syntax API is nothing more than a formal API that describes the familiar code constructs you know from C#. The full capabilities include information about how the code is formatted including line breaks, white space, and indenting. Using this information, you can fully represent the code as written and read by human programmers or the compiler. Using this structure enables you to interact with the source code on a deeply meaningful level. It's no longer text strings, but data that represents the structure of a C# program.

To get started, you'll need to install the **.NET Compiler Platform SDK**:

Installation instructions - Visual Studio Installer

There are two different ways to find the **.NET Compiler Platform SDK** in the **Visual Studio Installer**:

Install using the Visual Studio Installer - Workloads view

The **.NET Compiler Platform SDK** is not automatically selected as part of the Visual Studio extension

development workload. You must select it as an optional component.

1. Run **Visual Studio Installer**
2. Select **Modify**
3. Check the **Visual Studio extension development** workload.
4. Open the **Visual Studio extension development** node in the summary tree.
5. Check the box for **.NET Compiler Platform SDK**. You'll find it last under the optional components.

Optionally, you'll also want the **DGML editor** to display graphs in the visualizer:

1. Open the **Individual components** node in the summary tree.
2. Check the box for **DGML editor**

Install using the Visual Studio Installer - Individual components tab

1. Run **Visual Studio Installer**
2. Select **Modify**
3. Select the **Individual components** tab
4. Check the box for **.NET Compiler Platform SDK**. You'll find it at the top under the **Compilers, build tools, and runtimes** section.

Optionally, you'll also want the **DGML editor** to display graphs in the visualizer:

1. Check the box for **DGML editor**. You'll find it under the **Code tools** section.

Understanding syntax trees

You use the Syntax API for any analysis of the structure of C# code. The **Syntax API** exposes the parsers, the syntax trees, and utilities for analyzing and constructing syntax trees. It's how you search code for specific syntax elements or read the code for a program.

A syntax tree is a data structure used by the C# and Visual Basic compilers to understand C# and Visual Basic programs. Syntax trees are produced by the same parser that runs when a project is built or a developer hits F5. The syntax trees have full-fidelity with the language; every bit of information in a code file is represented in the tree. Writing a syntax tree to text reproduces the exact original text that was parsed. The syntax trees are also **immutable**; once created a syntax tree can never be changed. Consumers of the trees can analyze the trees on multiple threads, without locks or other concurrency measures, knowing the data never changes. You can use APIs to create new trees that are the result of modifying an existing tree.

The four primary building blocks of syntax trees are:

- The [Microsoft.CodeAnalysis.SyntaxTree](#) class, an instance of which represents an entire parse tree. [SyntaxTree](#) is an abstract class that has language-specific derivatives. You use the parse methods of the [Microsoft.CodeAnalysis.CSharp.CSharpSyntaxTree](#) (or [Microsoft.CodeAnalysis.VisualBasic.VisualBasicSyntaxTree](#)) class to parse text in C# (or Visual Basic).
- The [Microsoft.CodeAnalysis.SyntaxNode](#) class, instances of which represent syntactic constructs such as declarations, statements, clauses, and expressions.
- The [Microsoft.CodeAnalysis.SyntaxToken](#) structure, which represents an individual keyword, identifier, operator, or punctuation.
- And lastly the [Microsoft.CodeAnalysis.SyntaxTrivia](#) structure, which represents syntactically insignificant bits of information such as the white space between tokens, preprocessing directives, and comments.

Trivia, tokens, and nodes are composed hierarchically to form a tree that completely represents everything in a fragment of Visual Basic or C# code. You can see this structure using the **Syntax Visualizer** window. In Visual Studio, choose **View > Other Windows > Syntax Visualizer**. For example, the preceding C# source file examined using the **Syntax Visualizer** looks like the following figure:

SyntaxNode: Blue | SyntaxToken: Green | SyntaxTrivia: Red

The screenshot displays the Visual Studio IDE with a C# code file on the left and its Syntax Tree on the right. The code file, named 'Program', contains the following code:

```
using System;
using System.Collections.Generic;
using System.Linq;

class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine("Hello World");
    }
}
```

The Syntax Tree on the right shows the hierarchical structure of the code. The root node is 'CompilationUnit [0..185]', which branches into 'UsingDirective' nodes for 'System', 'System.Collections.Generic', and 'System.Linq'. The 'Program' class is represented by a 'ClassDeclaration' node, which contains a 'MethodDeclaration' for 'Main'. The 'Main' method's body is represented by a 'Block' node, which contains an 'ExpressionStatement' for the 'Console.WriteLine' call. The tree structure is as follows:

- CompilationUnit [0..185]
 - UsingDirective [0..13]
 - UsingDirective [15..48]
 - UsingDirective [50..68]
 - UsingKeyword [50..55]
 - QualifiedName [56..67]
 - IdentifierName [56..62]
 - DotToken [62..63]
 - IdentifierName [63..67]
 - IdentifierToken [63..67]
 - SemicolonToken [67..68]
 - Trail: EndOfLineTrivia [68..70]
 - ClassDeclaration [72..185]
 - ClassKeyword [72..77]
 - IdentifierToken [78..85]
 - OpenBraceToken [87..88]
 - Trail: EndOfLineTrivia [88..90]
 - MethodDeclaration [94..182]
 - StaticKeyword [94..100]
 - Lead: WhitespaceTrivia [90..94]
 - Trail: WhitespaceTrivia [100..101]
 - PredefinedType [101..105]
 - VoidKeyword [101..105]
 - IdentifierToken [106..110]
 - ParameterList [110..125]
 - OpenParenToken [110..111]
 - Parameter [111..124]
 - ArrayType [111..119]
 - PredefinedType [111..117]
 - ArrayRankSpecifier [117..119]
 - IdentifierToken [120..124]
 - CloseParenToken [124..125]
 - Block [131..182]
 - OpenBraceToken [131..132]
 - ExpressionStatement [142..175]
 - InvocationExpression [142..174]
 - SimpleMemberAccessExpression [142..159]
 - IdentifierName [142..149]
 - DotToken [149..150]
 - IdentifierName [150..159]
 - ArgumentList [159..174]
 - SemicolonToken [174..175]
 - CloseBraceToken [181..182]
 - CloseBraceToken [184..185]
 - EndOfFileToken [185..185]

By navigating this tree structure, you can find any statement, expression, token, or bit of white space in a code file.

While you can find anything in a code file using the Syntax APIs, most scenarios involve examining small snippets of code, or searching for particular statements or fragments. The two examples that follow show typical uses to browse the structure of code, or search for single statements.

Traversing trees

You can examine the nodes in a syntax tree in two ways. You can traverse the tree to examine each node, or you can query for specific elements or nodes.

Manual traversal

You can see the finished code for this sample in [our GitHub repository](#).

NOTE

The Syntax Tree types use inheritance to describe the different syntax elements that are valid at different locations in the program. Using these APIs often means casting properties or collection members to specific derived types. In the following examples, the assignment and the casts are separate statements, using explicitly typed variables. You can read the code to see the return types of the API and the runtime type of the objects returned. In practice, it's more common to use implicitly typed variables and rely on API names to describe the type of objects being examined.

Create a new C# **Stand-Alone Code Analysis Tool** project:

- In Visual Studio, choose **File > New > Project** to display the New Project dialog.
- Under **Visual C# > Extensibility**, choose **Stand-Alone Code Analysis Tool**.
- Name your project "SyntaxTreeManualTraversal" and click OK.

You're going to analyze the basic "Hello World!" program shown earlier. Add the text for the Hello World program as a constant in your `Program` class:

```
        const string programText =
@"using System;
using System.Collections;
using System.Linq;
using System.Text;

namespace HelloWorld
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Hello, World!");
        }
    }
}";
```

Next, add the following code to build the **syntax tree** for the code text in the `programText` constant. Add the following line to your `Main` method:

```
SyntaxTree tree = CSharpSyntaxTree.ParseText(programText);
CompilationUnitSyntax root = tree.GetCompilationUnitRoot();
```

Those two lines create the tree and retrieve the root node of that tree. You can now examine the nodes in the tree. Add these lines to your `Main` method to display some of the properties of the root node in the tree:

```
WriteLine($"The tree is a {root.Kind()} node.");
WriteLine($"The tree has {root.Members.Count} elements in it.");
WriteLine($"The tree has {root.Usings.Count} using statements. They are:");
foreach (UsingDirectiveSyntax element in root.Usings)
    WriteLine($"\\t{element.Name}");
```

Run the application to see what your code has discovered about the root node in this tree.

Typically, you'd traverse the tree to learn about the code. In this example, you're analyzing code you know to explore the APIs. Add the following code to examine the first member of the `root` node:

```
MemberDeclarationSyntax firstMember = root.Members[0];
WriteLine($"The first member is a {firstMember.Kind()}.");
var helloWorldDeclaration = (NamespaceDeclarationSyntax)firstMember;
```

That member is a [Microsoft.CodeAnalysis.CSharp.Syntax.NamespaceDeclarationSyntax](#). It represents everything in the scope of the `namespace HelloWorld` declaration. Add the following code to examine what nodes are declared inside the `HelloWorld` namespace:

```
WriteLine($"There are {helloWorldDeclaration.Members.Count} members declared in this namespace.");
WriteLine($"The first member is a {helloWorldDeclaration.Members[0].Kind()}.");
```

Run the program to see what you've learned.

Now that you know the declaration is a [Microsoft.CodeAnalysis.CSharp.Syntax.ClassDeclarationSyntax](#), declare a new variable of that type to examine the class declaration. This class only contains one member: the `Main` method. Add the following code to find the `Main` method, and cast it to a [Microsoft.CodeAnalysis.CSharp.Syntax.MethodDeclarationSyntax](#).

```
var programDeclaration = (ClassDeclarationSyntax)helloWorldDeclaration.Members[0];
WriteLine($"There are {programDeclaration.Members.Count} members declared in the
{programDeclaration.Identifier} class.");
WriteLine($"The first member is a {programDeclaration.Members[0].Kind()}.");
var mainDeclaration = (MethodDeclarationSyntax)programDeclaration.Members[0];
```

The method declaration node contains all the syntactic information about the method. Let's display the return type of the `Main` method, the number and types of the arguments, and the body text of the method. Add the following code:

```
WriteLine($"The return type of the {mainDeclaration.Identifier} method is {mainDeclaration.ReturnType}.");
WriteLine($"The method has {mainDeclaration.ParameterList.Parameters.Count} parameters.");
foreach (ParameterSyntax item in mainDeclaration.ParameterList.Parameters)
    WriteLine($"The type of the {item.Identifier} parameter is {item.Type}.");
WriteLine($"The body text of the {mainDeclaration.Identifier} method follows:");
WriteLine(mainDeclaration.Body.ToFullString());

var argsParameter = mainDeclaration.ParameterList.Parameters[0];
```

Run the program to see all the information you've discovered about this program:

```

The tree is a CompilationUnit node.
The tree has 1 elements in it.
The tree has 4 using statements. They are:
    System
    System.Collections
    System.Linq
    System.Text
The first member is a NamespaceDeclaration.
There are 1 members declared in this namespace.
The first member is a ClassDeclaration.
There are 1 members declared in the Program class.
The first member is a MethodDeclaration.
The return type of the Main method is void.
The method has 1 parameters.
The type of the args parameter is string[].
The body text of the Main method follows:
    {
        Console.WriteLine("Hello, World!");
    }

```

Query methods

In addition to traversing trees, you can also explore the syntax tree using the query methods defined on [Microsoft.CodeAnalysis.SyntaxNode](#). These methods should be immediately familiar to anyone familiar with XPath. You can use these methods with LINQ to quickly find things in a tree. The [SyntaxNode](#) has query methods such as [DescendantNodes](#), [AncestorsAndSelf](#) and [ChildNodes](#).

You can use these query methods to find the argument to the `Main` method as an alternative to navigating the tree. Add the following code to the bottom of your `Main` method:

```

var firstParameters = from methodDeclaration in root.DescendantNodes()
                      .OfType<MethodDeclarationSyntax>()
                      where methodDeclaration.Identifier.ValueText == "Main"
                      select methodDeclaration.ParameterList.Parameters.First();

var argsParameter2 = firstParameters.Single();

WriteLine(argsParameter == argsParameter2);

```

The first statement uses a LINQ expression and the [DescendantNodes](#) method to locate the same parameter as in the previous example.

Run the program, and you can see that the LINQ expression found the same parameter as manually navigating the tree.

The sample uses `WriteLine` statements to display information about the syntax trees as they are traversed. You can also learn much more by running the finished program under the debugger. You can examine more of the properties and methods that are part of the syntax tree created for the hello world program.

Syntax walkers

Often you want to find all nodes of a specific type in a syntax tree, for example, every property declaration in a file. By extending the [Microsoft.CodeAnalysis.CSharp.CSharpSyntaxWalker](#) class and overriding the [VisitPropertyDeclaration\(PropertyDeclarationSyntax\)](#) method, you process every property declaration in a syntax tree without knowing its structure beforehand. [CSharpSyntaxWalker](#) is a specific kind of [CSharpSyntaxVisitor](#) that recursively visits a node and each of its children.

This example implements a [CSharpSyntaxWalker](#) that examines a syntax tree. It collects `using` directives it finds that aren't importing a `System` namespace.

Create a new C# **Stand-Alone Code Analysis Tool** project; name it "SyntaxWalker."

You can see the finished code for this sample in [our GitHub repository](#). The sample on GitHub contains both projects described in this tutorial.

As in the previous sample, you can define a string constant to hold the text of the program you're going to analyze:

```
const string programText =
@"using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using Microsoft.CodeAnalysis;
using Microsoft.CodeAnalysis.CSharp;

namespace TopLevel
{
    using Microsoft;
    using System.ComponentModel;

    namespace Child1
    {
        using Microsoft.Win32;
        using System.Runtime.InteropServices;

        class Foo { }
    }

    namespace Child2
    {
        using System.CodeDom;
        using Microsoft.CSharp;

        class Bar { }
    }
}";
```

This source text contains `using` directives scattered across four different locations: the file-level, in the top-level namespace, and in the two nested namespaces. This example highlights a core scenario for using the [CSharpSyntaxWalker](#) class to query code. It would be cumbersome to visit every node in the root syntax tree to find using declarations. Instead, you create a derived class and override the method that gets called only when the current node in the tree is a using directive. Your visitor does not do any work on any other node types. This single method examines each of the `using` statements and builds a collection of the namespaces that aren't in the `System` namespace. You build a [CSharpSyntaxWalker](#) that examines all the `using` statements, but only the `using` statements.

Now that you've defined the program text, you need to create a `SyntaxTree` and get the root of that tree:

```
SyntaxTree tree = CSharpSyntaxTree.ParseText(programText);
CompilationUnitSyntax root = tree.GetCompilationUnitRoot();
```

Next, create a new class. In Visual Studio, choose **Project > Add New Item**. In the **Add New Item** dialog type *UsingCollector.cs* as the filename.

You implement the `using` visitor functionality in the `UsingCollector` class. Start by making the `UsingCollector` class derive from [CSharpSyntaxWalker](#).

```
class UsingCollector : CSharpSyntaxWalker
```


You need storage to hold the namespace nodes that you're collecting. Declare a public read-only property in the `UsingCollector` class; you use this variable to store the `UsingDirectiveSyntax` nodes you find:

```
public ICollection<UsingDirectiveSyntax> Usings { get; } = new List<UsingDirectiveSyntax>();
```

The base class, `CSharpSyntaxWalker` implements the logic to visit each node in the syntax tree. The derived class overrides the methods called for the specific nodes you're interested in. In this case, you're interested in any `using` directive. That means you must override the `VisitUsingDirective(UsingDirectiveSyntax)` method. The one argument to this method is a `Microsoft.CodeAnalysis.CSharp.Syntax.UsingDirectiveSyntax` object. That's an important advantage to using the visitors: they call the overridden methods with arguments already cast to the specific node type. The `Microsoft.CodeAnalysis.CSharp.Syntax.UsingDirectiveSyntax` class has a `Name` property that stores the name of the namespace being imported. It is a `Microsoft.CodeAnalysis.CSharp.Syntax.NameSyntax`. Add the following code in the `VisitUsingDirective(UsingDirectiveSyntax)` override:

```
public override void VisitUsingDirective(UsingDirectiveSyntax node)
{
    WriteLine($"\\tVisitUsingDirective called with {node.Name}.");
    if (node.Name.ToString() != "System" &&
        !node.Name.ToString().StartsWith("System."))
    {
        WriteLine($"\\t\\tSuccess. Adding {node.Name}.");
        this.Usings.Add(node);
    }
}
```

As with the earlier example, you've added a variety of `WriteLine` statements to aid in understanding of this method. You can see when it's called, and what arguments are passed to it each time.

Finally, you need to add two lines of code to create the `UsingCollector` and have it visit the root node, collecting all the `using` statements. Then, add a `foreach` loop to display all the `using` statements your collector found:

```
var collector = new UsingCollector();
collector.Visit(root);
foreach (var directive in collector.Usings)
{
    WriteLine(directive.Name);
}
```

Compile and run the program. You should see the following output:

```
VisitUsingDirective called with System.
VisitUsingDirective called with System.Collections.Generic.
VisitUsingDirective called with System.Linq.
VisitUsingDirective called with System.Text.
VisitUsingDirective called with Microsoft.CodeAnalysis.
    Success. Adding Microsoft.CodeAnalysis.
VisitUsingDirective called with Microsoft.CodeAnalysis.CSharp.
    Success. Adding Microsoft.CodeAnalysis.CSharp.
VisitUsingDirective called with Microsoft.
    Success. Adding Microsoft.
VisitUsingDirective called with System.ComponentModel.
VisitUsingDirective called with Microsoft.Win32.
    Success. Adding Microsoft.Win32.
VisitUsingDirective called with System.Runtime.InteropServices.
VisitUsingDirective called with System.CodeDom.
VisitUsingDirective called with Microsoft.CSharp.
    Success. Adding Microsoft.CSharp.
Microsoft.CodeAnalysis
Microsoft.CodeAnalysis.CSharp
Microsoft
Microsoft.Win32
Microsoft.CSharp
Press any key to continue . . .
```

Congratulations! You've used the **Syntax API** to locate specific kinds of C# statements and declarations in C# source code.

Get started with semantic analysis

12/28/2021 • 8 minutes to read • [Edit Online](#)

This tutorial assumes you're familiar with the Syntax API. The [get started with syntax analysis](#) article provides sufficient introduction.

In this tutorial, you explore the **Symbol** and **Binding** APIs. These APIs provide information about the *semantic meaning* of a program. They enable you to ask and answer questions about the types represented by any symbol in your program.

You'll need to install the .NET Compiler Platform SDK:

Installation instructions - Visual Studio Installer

There are two different ways to find the .NET Compiler Platform SDK in the Visual Studio Installer:

Install using the Visual Studio Installer - Workloads view

The .NET Compiler Platform SDK is not automatically selected as part of the Visual Studio extension development workload. You must select it as an optional component.

1. Run **Visual Studio Installer**
2. Select **Modify**
3. Check the **Visual Studio extension development** workload.
4. Open the **Visual Studio extension development** node in the summary tree.
5. Check the box for **.NET Compiler Platform SDK**. You'll find it last under the optional components.

Optionally, you'll also want the **DGML editor** to display graphs in the visualizer:

1. Open the **Individual components** node in the summary tree.
2. Check the box for **DGML editor**

Install using the Visual Studio Installer - Individual components tab

1. Run **Visual Studio Installer**
2. Select **Modify**
3. Select the **Individual components** tab
4. Check the box for **.NET Compiler Platform SDK**. You'll find it at the top under the **Compilers, build tools, and runtimes** section.

Optionally, you'll also want the **DGML editor** to display graphs in the visualizer:

1. Check the box for **DGML editor**. You'll find it under the **Code tools** section.

Understanding Compilations and Symbols

As you work more with the .NET Compiler SDK, you become familiar with the distinctions between Syntax API and the Semantic API. The **Syntax API** allows you to look at the *structure* of a program. However, often you want richer information about the semantics or *meaning* of a program. While a loose code file or snippet of Visual Basic or C# code can be syntactically analyzed in isolation, it's not meaningful to ask questions such as "what's the type of this variable" in a vacuum. The meaning of a type name may be dependent on assembly references, namespace imports, or other code files. Those questions are answered using the **Semantic API**, specifically the [Microsoft.CodeAnalysis.Compilation](#) class.

An instance of [Compilation](#) is analogous to a single project as seen by the compiler and represents everything needed to compile a Visual Basic or C# program. The **compilation** includes the set of source files to be compiled, assembly references, and compiler options. You can reason about the meaning of the code using all the other information in this context. A [Compilation](#) allows you to find **Symbols** - entities such as types, namespaces, members, and variables which names and other expressions refer to. The process of associating names and expressions with **Symbols** is called **Binding**.

Like [Microsoft.CodeAnalysis.SyntaxTree](#), [Compilation](#) is an abstract class with language-specific derivatives. When creating an instance of [Compilation](#), you must invoke a factory method on the [Microsoft.CodeAnalysis.CSharp.CSharpCompilation](#) (or [Microsoft.CodeAnalysis.VisualBasic.VisualBasicCompilation](#)) class.

Querying symbols

In this tutorial, you look at the "Hello World" program again. This time, you query the symbols in the program to understand what types those symbols represent. You query for the types in a namespace, and learn to find the methods available on a type.

You can see the finished code for this sample in [our GitHub repository](#).

NOTE

The Syntax Tree types use inheritance to describe the different syntax elements that are valid at different locations in the program. Using these APIs often means casting properties or collection members to specific derived types. In the following examples, the assignment and the casts are separate statements, using explicitly typed variables. You can read the code to see the return types of the API and the runtime type of the objects returned. In practice, it's more common to use implicitly typed variables and rely on API names to describe the type of objects being examined.

Create a new C# **Stand-Alone Code Analysis Tool** project:

- In Visual Studio, choose **File > New > Project** to display the New Project dialog.
- Under **Visual C# > Extensibility**, choose **Stand-Alone Code Analysis Tool**.
- Name your project "**SemanticQuickStart**" and click OK.

You're going to analyze the basic "Hello World!" program shown earlier. Add the text for the Hello World program as a constant in your `Program` class:

```
const string programText =
@"using System;
using System.Collections.Generic;
using System.Text;

namespace HelloWorld
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Hello, World!");
        }
    }
};
```

Next, add the following code to build the syntax tree for the code text in the `programText` constant. Add the following line to your `Main` method:

```
SyntaxTree tree = CSharpSyntaxTree.ParseText(programText);

CompilationUnitSyntax root = tree.GetCompilationUnitRoot();
```

Next, build a [CSharpCompilation](#) from the tree you already created. The "Hello World" sample relies on the [String](#) and [Console](#) types. You need to reference the assembly that declares those two types in your compilation. Add the following line to your `Main` method to create a compilation of your syntax tree, including the reference to the appropriate assembly:

```
var compilation = CSharpCompilation.Create("HelloWorld")
    .AddReferences(MetadataReference.CreateFromFile(
        typeof(string).Assembly.Location))
    .AddSyntaxTrees(tree);
```

The [CSharpCompilation.AddReferences](#) method adds references to the compilation. The [MetadataReference.CreateFromFile](#) method loads an assembly as a reference.

Querying the semantic model

Once you have a [Compilation](#) you can ask it for a [SemanticModel](#) for any [SyntaxTree](#) contained in that [Compilation](#). You can think of the semantic model as the source for all the information you would normally get from intellisense. A [SemanticModel](#) can answer questions like "What names are in scope at this location?", "What members are accessible from this method?", "What variables are used in this block of text?", and "What does this name/expression refer to?" Add this statement to create the semantic model:

```
SemanticModel model = compilation.GetSemanticModel(tree);
```

Binding a name

The [Compilation](#) creates the [SemanticModel](#) from the [SyntaxTree](#). After creating the model, you can query it to find the first `using` directive, and retrieve the symbol information for the `System` namespace. Add these two lines to your `Main` method to create the semantic model and retrieve the symbol for the first using statement:

```
// Use the syntax tree to find "using System;"
UsingDirectiveSyntax usingSystem = root.Usings[0];
NameSyntax systemName = usingSystem.Name;

// Use the semantic model for symbol information:
SymbolInfo nameInfo = model.GetSymbolInfo(systemName);
```

The preceding code shows how to bind the name in the first `using` directive to retrieve a [Microsoft.CodeAnalysis.SymbolInfo](#) for the `System` namespace. The preceding code also illustrates that you use the **syntax model** to find the structure of the code; you use the **semantic model** to understand its meaning. The **syntax model** finds the string `System` in the using statement. The **semantic model** has all the information about the types defined in the `System` namespace.

From the [SymbolInfo](#) object you can obtain the [Microsoft.CodeAnalysis.ISymbol](#) using the [SymbolInfo.Symbol](#) property. This property returns the symbol this expression refers to. For expressions that don't refer to anything (such as numeric literals) this property is `null`. When the [SymbolInfo.Symbol](#) is not null, the [ISymbol.Kind](#) denotes the type of the symbol. In this example, the [ISymbol.Kind](#) property is a [SymbolKind.Namespace](#). Add the following code to your `Main` method. It retrieves the symbol for the `System` namespace and then displays all the child namespaces declared in the `System` namespace:

```
var systemSymbol = (INamespaceSymbol)nameInfo.Symbol;
foreach (INamespaceSymbol ns in systemSymbol.GetNamespaceMembers())
{
    Console.WriteLine(ns);
}
```

Run the program and you should see the following output:

```
System.Collections
System.Configuration
System.Deployment
System.Diagnostics
System.Globalization
System.IO
System.Numerics
System.Reflection
System.Resources
System.Runtime
System.Security
System.StubHelpers
System.Text
System.Threading
Press any key to continue . . .
```

NOTE

The output does not include every namespace that is a child namespace of the `System` namespace. It displays every namespace that is present in this compilation, which only references the assembly where `System.String` is declared. Any namespaces declared in other assemblies are not known to this compilation

Binding an expression

The preceding code shows how to find a symbol by binding to a name. There are other expressions in a C# program that can be bound that aren't names. To demonstrate this capability, let's access the binding to a simple string literal.

The "Hello World" program contains a `Microsoft.CodeAnalysis.CSharp.Syntax.LiteralExpressionSyntax`, the "Hello, World!" string displayed to the console.

You find the "Hello, World!" string by locating the single string literal in the program. Then, once you've located the syntax node, get the type info for that node from the semantic model. Add the following code to your `Main` method:

```
// Use the syntax model to find the literal string:
LiteralExpressionSyntax helloWorldString = root.DescendantNodes()
    .OfType<LiteralExpressionSyntax>()
    .Single();

// Use the semantic model for type information:
TypeInfo literalInfo = model.GetTypeInfo(helloWorldString);
```

The `Microsoft.CodeAnalysis.TypeInfo` struct includes a `TypeInfo.Type` property that enables access to the semantic information about the type of the literal. In this example, that's the `string` type. Add a declaration that assigns this property to a local variable:

```
var stringTypeSymbol = (INamedTypeSymbol)literalInfo.Type;
```

To finish this tutorial, let's build a LINQ query that creates a sequence of all the public methods declared on the `string` type that return a `string`. This query gets complex, so let's build it line by line, then reconstruct it as a single query. The source for this query is the sequence of all members declared on the `string` type:

```
var allMembers = stringTypeSymbol.GetMembers();
```

That source sequence contains all members, including properties and fields, so filter it using the `ImmutableArray<T>.OfType` method to find elements that are `Microsoft.CodeAnalysis.IMethodSymbol` objects:

```
var methods = allMembers.OfType<IMethodSymbol>();
```

Next, add another filter to return only those methods that are public and return a `string`:

```
var publicStringReturningMethods = methods
    .Where(m => m.ReturnType.Equals(stringTypeSymbol) &&
        m.DeclaredAccessibility == Accessibility.Public);
```

Select only the name property, and only distinct names by removing any overloads:

```
var distinctMethods = publicStringReturningMethods.Select(m => m.Name).Distinct();
```

You can also build the full query using the LINQ query syntax, and then display all the method names in the console:

```
foreach (string name in (from method in stringTypeSymbol
    .GetMembers().OfType<IMethodSymbol>()
    where method.ReturnType.Equals(stringTypeSymbol) &&
        method.DeclaredAccessibility == Accessibility.Public
    select method.Name).Distinct())
{
    Console.WriteLine(name);
}
```

Build and run the program. You should see the following output:

```
Join
Substring
Trim
TrimStart
TrimEnd
Normalize
PadLeft
PadRight
ToLower
ToLowerInvariant
ToUpper
ToUpperInvariant
ToString
Insert
Replace
Remove
Format
Copy
Concat
Intern
IsInterned
Press any key to continue . . .
```

You've used the Semantic API to find and display information about the symbols that are part of this program.

Get started with syntax transformation

12/28/2021 • 12 minutes to read • [Edit Online](#)

This tutorial builds on concepts and techniques explored in the [Get started with syntax analysis](#) and [Get started with semantic analysis](#) quickstarts. If you haven't already, you should complete those quickstarts before beginning this one.

In this quickstart, you explore techniques for creating and transforming syntax trees. In combination with the techniques you learned in previous quickstarts, you create your first command-line refactoring!

Installation instructions - Visual Studio Installer

There are two different ways to find the .NET Compiler Platform SDK in the Visual Studio Installer:

Install using the Visual Studio Installer - Workloads view

The .NET Compiler Platform SDK is not automatically selected as part of the Visual Studio extension development workload. You must select it as an optional component.

1. Run **Visual Studio Installer**
2. Select **Modify**
3. Check the **Visual Studio extension development** workload.
4. Open the **Visual Studio extension development** node in the summary tree.
5. Check the box for **.NET Compiler Platform SDK**. You'll find it last under the optional components.

Optionally, you'll also want the **DGML editor** to display graphs in the visualizer:

1. Open the **Individual components** node in the summary tree.
2. Check the box for **DGML editor**

Install using the Visual Studio Installer - Individual components tab

1. Run **Visual Studio Installer**
2. Select **Modify**
3. Select the **Individual components** tab
4. Check the box for **.NET Compiler Platform SDK**. You'll find it at the top under the **Compilers, build tools, and runtimes** section.

Optionally, you'll also want the **DGML editor** to display graphs in the visualizer:

1. Check the box for **DGML editor**. You'll find it under the **Code tools** section.

Immutability and the .NET compiler platform

Immutability is a fundamental tenet of the .NET compiler platform. Immutable data structures can't be changed after they're created. Immutable data structures can be safely shared and analyzed by multiple consumers simultaneously. There's no danger that one consumer affects another in unpredictable ways. Your analyzer doesn't need locks or other concurrency measures. This rule applies to syntax trees, compilations, symbols, semantic models, and every other data structure you encounter. Instead of modifying existing structures, APIs create new objects based on specified differences to the old ones. You apply this concept to syntax trees to create new trees using transformations.

Create and transform trees

You choose one of two strategies for syntax transformations. **Factory methods** are best used when you're searching for specific nodes to replace, or specific locations where you want to insert new code. **Rewriters** are best when you want to scan an entire project for code patterns that you want to replace.

Create nodes with factory methods

The first syntax transformation demonstrates the factory methods. You're going to replace a `using System.Collections;` statement with a `using System.Collections.Generic;` statement. This example demonstrates how you create [Microsoft.CodeAnalysis.CSharp.CSharpSyntaxNode](#) objects using the [Microsoft.CodeAnalysis.CSharp.SyntaxFactory](#) factory methods. For each kind of **node**, **token**, or **trivia**, there's a factory method that creates an instance of that type. You create syntax trees by composing nodes hierarchically in a bottom-up fashion. Then, you'll transform the existing program by replacing existing nodes with the new tree you've created.

Start Visual Studio, and create a new **C# Stand-Alone Code Analysis Tool** project. In Visual Studio, choose **File > New > Project** to display the New Project dialog. Under **Visual C# > Extensibility** choose a **Stand-Alone Code Analysis Tool**. This quickstart has two example projects, so name the solution **SyntaxTransformationQuickStart**, and name the project **ConstructionCS**. Click OK.

This project uses the [Microsoft.CodeAnalysis.CSharp.SyntaxFactory](#) class methods to construct a [Microsoft.CodeAnalysis.CSharp.Syntax.NameSyntax](#) representing the `System.Collections.Generic` namespace.

Add the following using directive to the top of the `Program.cs`.

```
using static Microsoft.CodeAnalysis.CSharp.SyntaxFactory;
using static System.Console;
```

You'll create **name syntax nodes** to build the tree that represents the `using System.Collections.Generic;` statement. [NameSyntax](#) is the base class for four types of names that appear in C#. You compose these four types of names together to create any name that can appear in the C# language:

- [Microsoft.CodeAnalysis.CSharp.Syntax.NameSyntax](#), which represents simple single identifier names like `System` and `Microsoft`.
- [Microsoft.CodeAnalysis.CSharp.Syntax.GenericNameSyntax](#), which represents a generic type or method name such as `List<int>`.
- [Microsoft.CodeAnalysis.CSharp.Syntax.QualifiedNameSyntax](#), which represents a qualified name of the form `<left-name>.<right-identifier-or-generic-name>` such as `System.IO`.
- [Microsoft.CodeAnalysis.CSharp.Syntax.AliasQualifiedNameSyntax](#), which represents a name using an assembly extern alias such as `LibraryV2::Foo`.

You use the [IdentifierName\(String\)](#) method to create a [NameSyntax](#) node. Add the following code in your `Main` method in `Program.cs`:

```
NameSyntax name = IdentifierName("System");
WriteLine($"{name}\tCreated the identifier {name}");
```

The preceding code creates an [IdentifierNameSyntax](#) object and assigns it to the variable `name`. Many of the Roslyn APIs return base classes to make it easier to work with related types. The variable `name`, an [NameSyntax](#), can be reused as you build the [QualifiedNameSyntax](#). Don't use type inference as you build the sample. You'll automate that step in this project.

You've created the name. Now, it's time to build more nodes into the tree by building a [QualifiedNameSyntax](#). The new tree uses `name` as the left of the name, and a new [IdentifierNameSyntax](#) for the `Collections`

namespace as the right side of the [QualifiedNameSyntax](#). Add the following code to `program.cs`:

```
name = QualifiedName(name, IdentifierName("Collections"));
WriteLine(name.ToString());
```

Run the code again, and see the results. You're building a tree of nodes that represents code. You'll continue this pattern to build the [QualifiedNameSyntax](#) for the namespace `System.Collections.Generic`. Add the following code to `Program.cs`:

```
name = QualifiedName(name, IdentifierName("Generic"));
WriteLine(name.ToString());
```

Run the program again to see that you've built the tree for the code to add.

Create a modified tree

You've built a small syntax tree that contains one statement. The APIs to create new nodes are the right choice to create single statements or other small code blocks. However, to build larger blocks of code, you should use methods that replace nodes or insert nodes into an existing tree. Remember that syntax trees are immutable. The **Syntax API** doesn't provide any mechanism for modifying an existing syntax tree after construction. Instead, it provides methods that produce new trees based on changes to existing ones. `With*` methods are defined in concrete classes that derive from [SyntaxNode](#) or in extension methods declared in the [SyntaxNodeExtensions](#) class. These methods create a new node by applying changes to an existing node's child properties. Additionally, the [ReplaceNode](#) extension method can be used to replace a descendent node in a subtree. This method also updates the parent to point to the newly created child and repeats this process up the entire tree - a process known as *re-spinning* the tree.

The next step is to create a tree that represents an entire (small) program and then modify it. Add the following code to the beginning of the `Program` class:

```
private const string sampleCode =
@"using System;
using System.Collections;
using System.Linq;
using System.Text;

namespace HelloWorld
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Hello, World!");
        }
    }
}";
```

NOTE

The example code uses the `System.Collections` namespace and not the `System.Collections.Generic` namespace.

Next, add the following code to the bottom of the `Main` method to parse the text and create a tree:

```
SyntaxTree tree = CSharpSyntaxTree.ParseText(sampleCode);
var root = (CompilationUnitSyntax)tree.GetRoot();
```

This example uses the [WithName\(NameSyntax\)](#) method to replace the name in a [UsingDirectiveSyntax](#) node with the one constructed in the preceding code.

Create a new [UsingDirectiveSyntax](#) node using the [WithName\(NameSyntax\)](#) method to update the `System.Collections` name with the name you created in the preceding code. Add the following code to the bottom of the `Main` method:

```
var oldUsing = root.Usings[1];
var newUsing = oldUsing.WithName(name);
WriteLine(root.ToString());
```

Run the program and look carefully at the output. The `newUsing` hasn't been placed in the root tree. The original tree hasn't been changed.

Add the following code using the [ReplaceNode](#) extension method to create a new tree. The new tree is the result of replacing the existing import with the updated `newUsing` node. You assign this new tree to the existing `root` variable:

```
root = root.ReplaceNode(oldUsing, newUsing);
WriteLine(root.ToString());
```

Run the program again. This time the tree now correctly imports the `System.Collections.Generic` namespace.

Transform trees using `SyntaxRewriters`

The `With*` and [ReplaceNode](#) methods provide convenient means to transform individual branches of a syntax tree. The [Microsoft.CodeAnalysis.CSharp.CSharpSyntaxRewriter](#) class performs multiple transformations on a syntax tree. The [Microsoft.CodeAnalysis.CSharp.CSharpSyntaxRewriter](#) class is a subclass of [Microsoft.CodeAnalysis.CSharp.CSharpSyntaxVisitor<TResult>](#). The [CSharpSyntaxRewriter](#) applies a transformation to a specific type of [SyntaxNode](#). You can apply transformations to multiple types of [SyntaxNode](#) objects wherever they appear in a syntax tree. The second project in this quickstart creates a command-line refactoring that removes explicit types in local variable declarations anywhere that type inference could be used.

Create a new **C# Stand-Alone Code Analysis Tool** project. In Visual Studio, right-click the `SyntaxTransformationQuickStart` solution node. Choose **Add > New Project** to display the **New Project** dialog. Under **Visual C# > Extensibility**, choose **Stand-Alone Code Analysis Tool**. Name your project `TransformationCS` and click OK.

The first step is to create a class that derives from [CSharpSyntaxRewriter](#) to perform your transformations. Add a new class file to the project. In Visual Studio, choose **Project > Add Class....** In the **Add New Item** dialog type `TypeInferenceRewriter.cs` as the filename.

Add the following using directives to the `TypeInferenceRewriter.cs` file:

```
using Microsoft.CodeAnalysis;
using Microsoft.CodeAnalysis.CSharp;
using Microsoft.CodeAnalysis.CSharp.Syntax;
```

Next, make the `TypeInferenceRewriter` class extend the [CSharpSyntaxRewriter](#) class:

```
public class TypeInferenceRewriter : CSharpSyntaxRewriter
```

Add the following code to declare a private read-only field to hold a [SemanticModel](#) and initialize it in the constructor. You will need this field later on to determine where type inference can be used:

```
private readonly SemanticModel SemanticModel;

public TypeInferenceRewriter(SemanticModel semanticModel) => SemanticModel = semanticModel;
```

Override the [VisitLocalDeclarationStatement\(LocalDeclarationStatementSyntax\)](#) method:

```
public override SyntaxNode VisitLocalDeclarationStatement(LocalDeclarationStatementSyntax node)
{
}
}
```

NOTE

Many of the Roslyn APIs declare return types that are base classes of the actual runtime types returned. In many scenarios, one kind of node may be replaced by another kind of node entirely - or even removed. In this example, the [VisitLocalDeclarationStatement\(LocalDeclarationStatementSyntax\)](#) method returns a [SyntaxNode](#), instead of the derived type of [LocalDeclarationStatementSyntax](#). This rewriter returns a new [LocalDeclarationStatementSyntax](#) node based on the existing one.

This quickstart handles local variable declarations. You could extend it to other declarations such as `foreach` loops, `for` loops, LINQ expressions, and lambda expressions. Furthermore this rewriter will only transform declarations of the simplest form:

```
Type variable = expression;
```

If you want to explore on your own, consider extending the finished sample for these types of variable declarations:

```
// Multiple variables in a single declaration.
Type variable1 = expression1,
    variable2 = expression2;
// No initializer.
Type variable;
```

Add the following code to the body of the [VisitLocalDeclarationStatement](#) method to skip rewriting these forms of declarations:

```
if (node.Declaration.Variables.Count > 1)
{
    return node;
}
if (node.Declaration.Variables[0].Initializer == null)
{
    return node;
}
```

The method indicates that no rewriting takes place by returning the `node` parameter unmodified. If neither of those `if` expressions are true, the node represents a possible declaration with initialization. Add these statements to extract the type name specified in the declaration and bind it using the [SemanticModel](#) field to obtain a type symbol:

```
var declarator = node.Declaration.Variables.First();
var variableTypeName = node.Declaration.Type;

var variableType = (ITypeSymbol)SemanticModel
    .GetSymbolInfo(variableTypeName)
    .Symbol;
```

Now, add this statement to bind the initializer expression:

```
var initializerInfo = SemanticModel.GetTypeInfo(declarator.Initializer.Value);
```

Finally, add the following `if` statement to replace the existing type name with the `var` keyword if the type of the initializer expression matches the type specified:

```
if (SymbolEqualityComparer.Default.Equals(variableType, initializerInfo.Type))
{
    TypeSyntax varTypeName = SyntaxFactory.IdentifierName("var")
        .WithLeadingTrivia(variableTypeName.GetLeadingTrivia())
        .WithTrailingTrivia(variableTypeName.GetTrailingTrivia());

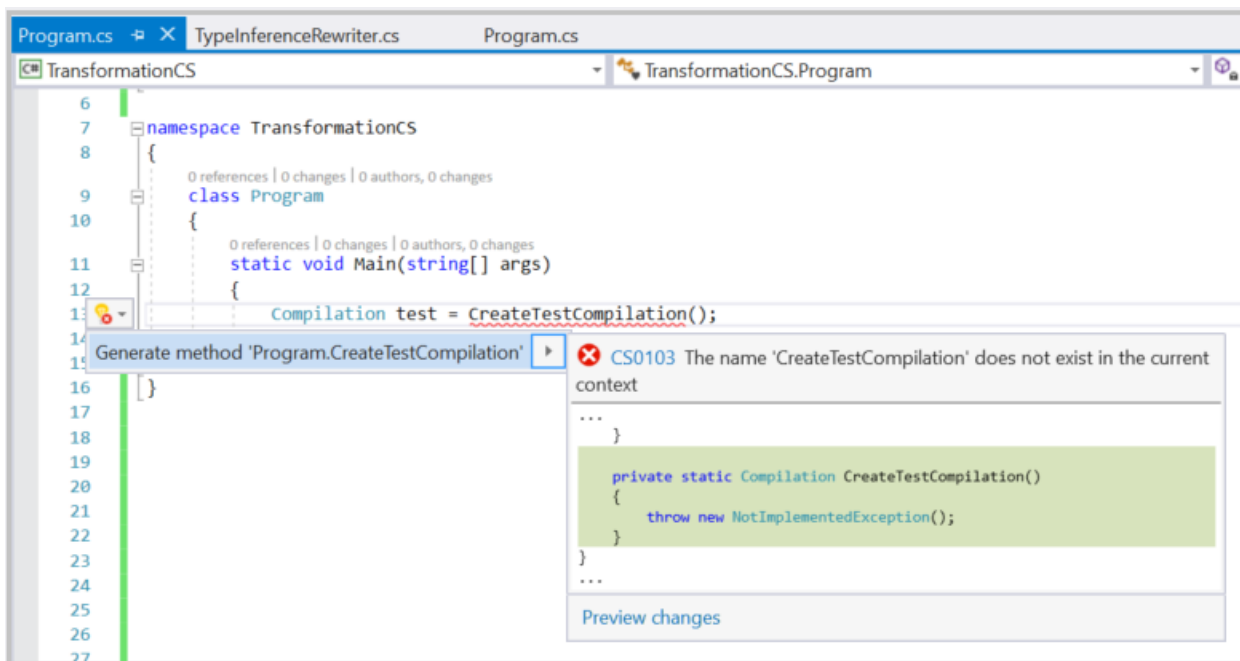
    return node.ReplaceNode(variableTypeName, varTypeName);
}
else
{
    return node;
}
```

The conditional is required because the declaration may cast the initializer expression to a base class or interface. If that's desired, the types on the left and right-hand side of the assignment don't match. Removing the explicit type in these cases would change the semantics of a program. `var` is specified as an identifier rather than a keyword because `var` is a contextual keyword. The leading and trailing trivia (white space) are transferred from the old type name to the `var` keyword to maintain vertical white space and indentation. It's simpler to use `ReplaceNode` rather than `With*` to transform the [LocalDeclarationStatementSyntax](#) because the type name is actually the grandchild of the declaration statement.

You've finished the `TypeInferenceRewriter`. Now return to your `Program.cs` file to finish the example. Create a test [Compilation](#) and obtain the [SemanticModel](#) from it. Use that [SemanticModel](#) to try your `TypeInferenceRewriter`. You'll do this step last. In the meantime declare a placeholder variable representing your test compilation:

```
Compilation test = CreateTestCompilation();
```

After pausing a moment, you should see an error squiggle appear reporting that no `CreateTestCompilation` method exists. Press **Ctrl+Period** to open the light-bulb and then press Enter to invoke the **Generate Method Stub** command. This command will generate a method stub for the `CreateTestCompilation` method in the `Program` class. You'll come back to fill in this method later:



Write the following code to iterate over each [SyntaxTree](#) in the test [Compilation](#). For each one, initialize a new `TypeInferenceRewriter` with the [SemanticModel](#) for that tree:

```
foreach (SyntaxTree sourceTree in test.SyntaxTrees)
{
    SemanticModel model = test.GetSemanticModel(sourceTree);

    TypeInferenceRewriter rewriter = new TypeInferenceRewriter(model);

    SyntaxNode newSource = rewriter.Visit(sourceTree.GetRoot());

    if (newSource != sourceTree.GetRoot())
    {
        File.WriteAllText(sourceTree.FilePath, newSource.ToFullString());
    }
}
```

Inside the `foreach` statement you created, add the following code to perform the transformation on each source tree. This code conditionally writes out the new transformed tree if any edits were made. Your rewriter should only modify a tree if it encounters one or more local variable declarations that could be simplified using type inference:

```
SyntaxNode newSource = rewriter.Visit(sourceTree.GetRoot());

if (newSource != sourceTree.GetRoot())
{
    File.WriteAllText(sourceTree.FilePath, newSource.ToFullString());
}
```

You should see squiggles under the `File.WriteAllText` code. Select the light bulb, and add the necessary `using System.IO;` statement.

You're almost done! There's one step left: creating a test [Compilation](#). Since you haven't been using type inference at all during this quickstart, it would have made a perfect test case. Unfortunately, creating a `Compilation` from a C# project file is beyond the scope of this walkthrough. But fortunately, if you've been following instructions carefully, there's hope. Replace the contents of the `CreateTestCompilation` method with the following code. It creates a test compilation that coincidentally matches the project described in this quickstart:

```

String programPath = @"..\..\..\Program.cs";
String programText = File.ReadAllText(programPath);
SyntaxTree programTree =
    CSharpSyntaxTree.ParseText(programText)
        .WithFilePath(programPath);

String rewriterPath = @"..\..\..\TypeInferenceRewriter.cs";
String rewriterText = File.ReadAllText(rewriterPath);
SyntaxTree rewriterTree =
    CSharpSyntaxTree.ParseText(rewriterText)
        .WithFilePath(rewriterPath);

SyntaxTree[] sourceTrees = { programTree, rewriterTree };

MetadataReference mscorlib =
    MetadataReference.CreateFromFile(typeof(object).Assembly.Location);
MetadataReference codeAnalysis =
    MetadataReference.CreateFromFile(typeof(SyntaxTree).Assembly.Location);
MetadataReference csharpCodeAnalysis =
    MetadataReference.CreateFromFile(typeof(CSharpSyntaxTree).Assembly.Location);

MetadataReference[] references = { mscorlib, codeAnalysis, csharpCodeAnalysis };

return CSharpCompilation.Create("TransformationCS",
    sourceTrees,
    references,
    new CSharpCompilationOptions(OutputKind.ConsoleApplication));

```

Cross your fingers and run the project. In Visual Studio, choose **Debug > Start Debugging**. You should be prompted by Visual Studio that the files in your project have changed. Click "**Yes to All**" to reload the modified files. Examine them to observe your awesomeness. Note how much cleaner the code looks without all those explicit and redundant type specifiers.

Congratulations! You've used the **Compiler APIs** to write your own refactoring that searches all files in a C# project for certain syntactic patterns, analyzes the semantics of source code that matches those patterns, and transforms it. You're now officially a refactoring author!

Tutorial: Write your first analyzer and code fix

12/28/2021 • 24 minutes to read • [Edit Online](#)

The .NET Compiler Platform SDK provides the tools you need to create custom diagnostics (analyzers), code fixes, code refactoring, and diagnostic suppressors that target C# or Visual Basic code. An **analyzer** contains code that recognizes violations of your rule. Your **code fix** contains the code that fixes the violation. The rules you implement can be anything from code structure to coding style to naming conventions and more. The .NET Compiler Platform provides the framework for running analysis as developers are writing code, and all the Visual Studio UI features for fixing code: showing squiggles in the editor, populating the Visual Studio Error List, creating the "light bulb" suggestions and showing the rich preview of the suggested fixes.

In this tutorial, you'll explore the creation of an **analyzer** and an accompanying **code fix** using the Roslyn APIs. An analyzer is a way to perform source code analysis and report a problem to the user. Optionally, a code fix can be associated with the analyzer to represent a modification to the user's source code. This tutorial creates an analyzer that finds local variable declarations that could be declared using the `const` modifier but are not. The accompanying code fix modifies those declarations to add the `const` modifier.

Prerequisites

- [Visual Studio 2019](#) version 16.8 or later

You'll need to install the .NET Compiler Platform SDK via the Visual Studio Installer:

Installation instructions - Visual Studio Installer

There are two different ways to find the .NET Compiler Platform SDK in the Visual Studio Installer:

Install using the Visual Studio Installer - Workloads view

The .NET Compiler Platform SDK is not automatically selected as part of the Visual Studio extension development workload. You must select it as an optional component.

1. Run **Visual Studio Installer**
2. Select **Modify**
3. Check the **Visual Studio extension development** workload.
4. Open the **Visual Studio extension development** node in the summary tree.
5. Check the box for **.NET Compiler Platform SDK**. You'll find it last under the optional components.

Optionally, you'll also want the **DGML editor** to display graphs in the visualizer:

1. Open the **Individual components** node in the summary tree.
2. Check the box for **DGML editor**

Install using the Visual Studio Installer - Individual components tab

1. Run **Visual Studio Installer**
2. Select **Modify**
3. Select the **Individual components** tab
4. Check the box for **.NET Compiler Platform SDK**. You'll find it at the top under the **Compilers, build tools, and runtimes** section.

Optionally, you'll also want the **DGML editor** to display graphs in the visualizer:

1. Check the box for **DGML editor**. You'll find it under the **Code tools** section.

There are several steps to creating and validating your analyzer:

1. Create the solution.
2. Register the analyzer name and description.
3. Report analyzer warnings and recommendations.
4. Implement the code fix to accept recommendations.
5. Improve the analysis through unit tests.

Create the solution

- In Visual Studio, choose **File > New > Project...** to display the New Project dialog.
- Under **Visual C# > Extensibility**, choose **Analyzer with code fix (.NET Standard)**.
- Name your project "**MakeConst**" and click OK.

Explore the analyzer template

The analyzer with code fix template creates five projects:

- **MakeConst**, which contains the analyzer.
- **MakeConst.CodeFixes**, which contains the code fix.
- **MakeConst.Package**, which is used to produce NuGet package for the analyzer and code fix.
- **MakeConst.Test**, which is a unit test project.
- **MakeConst.Vsix**, which is the default startup project that starts a second instance of Visual Studio that has loaded your new analyzer. Press F5 to start the VSIX project.

NOTE

Analyzers should target .NET Standard 2.0 because they can run in .NET Core environment (command line builds) and .NET Framework environment (Visual Studio).

TIP

When you run your analyzer, you start a second copy of Visual Studio. This second copy uses a different registry hive to store settings. That enables you to differentiate the visual settings in the two copies of Visual Studio. You can pick a different theme for the experimental run of Visual Studio. In addition, don't roam your settings or login to your Visual Studio account using the experimental run of Visual Studio. That keeps the settings different.

The hive includes not only the analyzer under development, but also any previous analyzers opened. To reset Roslyn hive, you need to manually delete it from `%LocalAppData%\Microsoft\VisualStudio`. The folder name of Roslyn hive will end in `Roslyn`, for example, `16.0_9ae182f9Roslyn`. Note that you may need to clean the solution and rebuild it after deleting the hive.

In the second Visual Studio instance that you just started, create a new C# Console Application project (any target framework will work -- analyzers work at the source level.) Hover over the token with a wavy underline, and the warning text provided by an analyzer appears.

The template creates an analyzer that reports a warning on each type declaration where the type name contains lowercase letters, as shown in the following figure:

```

namespace ConsoleApp1
{
    0 references
    class Program
    {
        0 references
        static void Main(string[] args)
        {
            Console.WriteLine("Hello World!");
        }
    }
}

```

The template also provides a code fix that changes any type name containing lower case characters to all upper case. You can click on the light bulb displayed with the warning to see the suggested changes. Accepting the suggested changes updates the type name and all references to that type in the solution. Now that you've seen the initial analyzer in action, close the second Visual Studio instance and return to your analyzer project.

You don't have to start a second copy of Visual Studio and create new code to test every change in your analyzer. The template also creates a unit test project for you. That project contains two tests. `TestMethod1` shows the typical format of a test that analyzes code without triggering a diagnostic. `TestMethod2` shows the format of a test that triggers a diagnostic, and then applies a suggested code fix. As you build your analyzer and code fix, you'll write tests for different code structures to verify your work. Unit tests for analyzers are much quicker than testing them interactively with Visual Studio.

TIP

Analyzer unit tests are a great tool when you know what code constructs should and shouldn't trigger your analyzer. Loading your analyzer in another copy of Visual Studio is a great tool to explore and find constructs you may not have thought about yet.

In this tutorial, you write an analyzer that reports to the user any local variable declarations that can be converted to local constants. For example, consider the following code:

```

int x = 0;
Console.WriteLine(x);

```

In the code above, `x` is assigned a constant value and is never modified. It can be declared using the `const` modifier:

```

const int x = 0;
Console.WriteLine(x);

```

The analysis to determine whether a variable can be made constant is involved, requiring syntactic analysis, constant analysis of the initializer expression and dataflow analysis to ensure that the variable is never written to. The .NET Compiler Platform provides APIs that make it easier to perform this analysis.

Create analyzer registrations

The template creates the initial `DiagnosticAnalyzer` class, in the `MakeConstAnalyzer.cs` file. This initial analyzer shows two important properties of every analyzer.

- Every diagnostic analyzer must provide a `[DiagnosticAnalyzer]` attribute that describes the language it operates on.
- Every diagnostic analyzer must derive (directly or indirectly) from the `DiagnosticAnalyzer` class.

The template also shows the basic features that are part of any analyzer:

1. Register actions. The actions represent code changes that should trigger your analyzer to examine code for violations. When Visual Studio detects code edits that match a registered action, it calls your analyzer's registered method.
2. Create diagnostics. When your analyzer detects a violation, it creates a diagnostic object that Visual Studio uses to notify the user of the violation.

You register actions in your override of `DiagnosticAnalyzer.Initialize(AnalysisContext)` method. In this tutorial, you'll visit **syntax nodes** looking for local declarations, and see which of those have constant values. If a declaration could be constant, your analyzer will create and report a diagnostic.

The first step is to update the registration constants and `Initialize` method so these constants indicate your "Make Const" analyzer. Most of the string constants are defined in the string resource file. You should follow that practice for easier localization. Open the *Resources.resx* file for the **MakeConst** analyzer project. This displays the resource editor. Update the string resources as follows:

- Change `AnalyzerDescription` to "Variables that are not modified should be made constants."
- Change `AnalyzerMessageFormat` to "Variable '{0}' can be made constant".
- Change `AnalyzerTitle` to "Variable can be made constant".

When you have finished, the resource editor should appear as shown in the following figure:

	Name	Value	Comment
	AnalyzerDescription	Variables that are not modified should be made constants.	An optional longer localizable description of the diagnostic.
	AnalyzerMessageFormat	Variable '{0}' can be made constant	The format-able message the diagnostic displays.
▶	AnalyzerTitle	Variable can be made constant	The title of the diagnostic.
*			

The remaining changes are in the analyzer file. Open *MakeConstAnalyzer.cs* in Visual Studio. Change the registered action from one that acts on symbols to one that acts on syntax. In the

`MakeConstAnalyzerAnalyzer.Initialize` method, find the line that registers the action on symbols:

```
context.RegisterSymbolAction(AnalyzeSymbol, SymbolKind.NamedType);
```

Replace it with the following line:

```
context.RegisterSyntaxNodeAction(AnalyzeNode, SyntaxKind.LocalDeclarationStatement);
```

After that change, you can delete the `AnalyzeSymbol` method. This analyzer examines `SyntaxKind.LocalDeclarationStatement`, not `SymbolKind.NamedType` statements. Notice that `AnalyzeNode` has red squiggles under it. The code you just added references an `AnalyzeNode` method that hasn't been declared. Declare that method using the following code:

```
private void AnalyzeNode(SyntaxNodeAnalysisContext context)
{
}
```

Change the `Category` to "Usage" in *MakeConstAnalyzer.cs* as shown in the following code:

```
private const string Category = "Usage";
```

Find local declarations that could be const

It's time to write the first version of the `AnalyzeNode` method. It should look for a single local declaration that could be `const` but is not, like the following code:

```
int x = 0;
Console.WriteLine(x);
```

The first step is to find local declarations. Add the following code to `AnalyzeNode` in *MakeConstAnalyzer.cs*.

```
var localDeclaration = (LocalDeclarationStatementSyntax)context.Node;
```

This cast always succeeds because your analyzer registered for changes to local declarations, and only local declarations. No other node type triggers a call to your `AnalyzeNode` method. Next, check the declaration for any `const` modifiers. If you find them, return immediately. The following code looks for any `const` modifiers on the local declaration:

```
// make sure the declaration isn't already const:
if (localDeclaration.Modifiers.Any(SyntaxKind.ConstKeyword))
{
    return;
}
```

Finally, you need to check that the variable could be `const`. That means making sure it is never assigned after it is initialized.

You'll perform some semantic analysis using the [SyntaxNodeAnalysisContext](#). You use the `context` argument to determine whether the local variable declaration can be made `const`. A [Microsoft.CodeAnalysis.SemanticModel](#) represents all of semantic information in a single source file. You can learn more in the article that covers [semantic models](#). You'll use the [Microsoft.CodeAnalysis.SemanticModel](#) to perform data flow analysis on the local declaration statement. Then, you use the results of this data flow analysis to ensure that the local variable isn't written with a new value anywhere else. Call the [GetDeclaredSymbol](#) extension method to retrieve the [ILocalSymbol](#) for the variable and check that it isn't contained with the [DataFlowAnalysis.WrittenOutside](#) collection of the data flow analysis. Add the following code to the end of the `AnalyzeNode` method:

```
// Perform data flow analysis on the local declaration.
DataFlowAnalysis dataFlowAnalysis = context.SemanticModel.AnalyzeDataFlow(localDeclaration);

// Retrieve the local symbol for each variable in the local declaration
// and ensure that it is not written outside of the data flow analysis region.
VariableDeclaratorSyntax variable = localDeclaration.Declaration.Variables.Single();
ISymbol variableSymbol = context.SemanticModel.GetDeclaredSymbol(variable, context.CancellationToken);
if (dataFlowAnalysis.WrittenOutside.Contains(variableSymbol))
{
    return;
}
```

The code just added ensures that the variable isn't modified, and can therefore be made `const`. It's time to raise the diagnostic. Add the following code as the last line in `AnalyzeNode`:

```
context.ReportDiagnostic(Diagnostic.Create(Rule, context.Node.GetLocation(),
localDeclaration.Declaration.Variables.First().Identifier.ValueText));
```

You can check your progress by pressing F5 to run your analyzer. You can load the console application you

created earlier and then add the following test code:

```
int x = 0;
Console.WriteLine(x);
```

The light bulb should appear, and your analyzer should report a diagnostic. However, the light bulb still uses the template generated code fix, and tells you it can be made upper case. The next section explains how to write the code fix.

Write the code fix

An analyzer can provide one or more code fixes. A code fix defines an edit that addresses the reported issue. For the analyzer that you created, you can provide a code fix that inserts the `const` keyword:

```
- int x = 0;
+ const int x = 0;
Console.WriteLine(x);
```

The user chooses it from the light bulb UI in the editor and Visual Studio changes the code.

Open *CodeFixResources.resx* file and change `CodeFixTitle` to "Make constant".

Open the *MakeConstCodeFixProvider.cs* file added by the template. This code fix is already wired up to the Diagnostic ID produced by your diagnostic analyzer, but it doesn't yet implement the right code transform.

Next, delete the `MakeUppercaseAsync` method. It no longer applies.

All code fix providers derive from [CodeFixProvider](#). They all override [CodeFixProvider.RegisterCodeFixesAsync\(CodeFixContext\)](#) to report available code fixes. In `RegisterCodeFixesAsync`, change the ancestor node type you're searching for to a [LocalDeclarationStatementSyntax](#) to match the diagnostic:

```
var declaration =
root.FindToken(diagnosticSpan.Start).Parent.AncestorsAndSelf().OfType<LocalDeclarationStatementSyntax>
().First();
```

Next, change the last line to register a code fix. Your fix will create a new document that results from adding the `const` modifier to an existing declaration:

```
// Register a code action that will invoke the fix.
context.RegisterCodeFix(
    CodeAction.Create(
        title: CodeFixResources.CodeFixTitle,
        createChangedDocument: c => MakeConstAsync(context.Document, declaration, c),
        equivalenceKey: nameof(CodeFixResources.CodeFixTitle)),
    diagnostic);
```

You'll notice red squiggles in the code you just added on the symbol `MakeConstAsync`. Add a declaration for `MakeConstAsync` like the following code:

```
private static async Task<Document> MakeConstAsync(Document document,
    LocalDeclarationStatementSyntax localDeclaration,
    CancellationToken cancellationToken)
{
}
```

Your new `MakeConstAsync` method will transform the `Document` representing the user's source file into a new `Document` that now contains a `const` declaration.

You create a new `const` keyword token to insert at the front of the declaration statement. Be careful to first remove any leading trivia from the first token of the declaration statement and attach it to the `const` token. Add the following code to the `MakeConstAsync` method:

```
// Remove the leading trivia from the local declaration.
SyntaxToken firstToken = localDeclaration.GetFirstToken();
SyntaxTriviaList leadingTrivia = firstToken.LeadingTrivia;
LocalDeclarationStatementSyntax trimmedLocal = localDeclaration.ReplaceToken(
    firstToken, firstToken.WithLeadingTrivia(SyntaxTriviaList.Empty));

// Create a const token with the leading trivia.
SyntaxToken constToken = SyntaxFactory.Token(leadingTrivia, SyntaxKind.ConstKeyword,
    SyntaxFactory.TriviaList(SyntaxFactory.ElasticMarker));
```

Next, add the `const` token to the declaration using the following code:

```
// Insert the const token into the modifiers list, creating a new modifiers list.
SyntaxTokenList newModifiers = trimmedLocal.Modifiers.Insert(0, constToken);
// Produce the new local declaration.
LocalDeclarationStatementSyntax newLocal = trimmedLocal
    .WithModifiers(newModifiers)
    .WithDeclaration(localDeclaration.Declaration);
```

Next, format the new declaration to match C# formatting rules. Formatting your changes to match existing code creates a better experience. Add the following statement immediately after the existing code:

```
// Add an annotation to format the new local declaration.
LocalDeclarationStatementSyntax formattedLocal = newLocal.WithAdditionalAnnotations(Formatter.Annotation);
```

A new namespace is required for this code. Add the following `using` directive to the top of the file:

```
using Microsoft.CodeAnalysis.Formatting;
```

The final step is to make your edit. There are three steps to this process:

1. Get a handle to the existing document.
2. Create a new document by replacing the existing declaration with the new declaration.
3. Return the new document.

Add the following code to the end of the `MakeConstAsync` method:

```
// Replace the old local declaration with the new local declaration.
SyntaxNode oldRoot = await document.GetSyntaxRootAsync(cancellationToken).ConfigureAwait(false);
SyntaxNode newRoot = oldRoot.ReplaceNode(localDeclaration, formattedLocal);

// Return document with transformed tree.
return document.WithSyntaxRoot(newRoot);
```

Your code fix is ready to try. Press F5 to run the analyzer project in a second instance of Visual Studio. In the second Visual Studio instance, create a new C# Console Application project and add a few local variable declarations initialized with constant values to the Main method. You'll see that they are reported as warnings as below.

```
static void Main(string[] args)
{
    int i = 1;
    int j = 2;
    int k = i + j;
}
```

You've made a lot of progress. There are squiggles under the declarations that can be made `const`. But there is still work to do. This works fine if you add `const` to the declarations starting with `i`, then `j` and finally `k`. But, if you add the `const` modifier in a different order, starting with `k`, your analyzer creates errors: `k` can't be declared `const`, unless `i` and `j` are both already `const`. You've got to do more analysis to ensure you handle the different ways variables can be declared and initialized.

Build unit tests

Your analyzer and code fix work on a simple case of a single declaration that can be made `const`. There are numerous possible declaration statements where this implementation makes mistakes. You'll address these cases by working with the unit test library written by the template. It's much faster than repeatedly opening a second copy of Visual Studio.

Open the *MakeConstUnitTests.cs* file in the unit test project. The template created two tests that follow the two common patterns for an analyzer and code fix unit test. `TestMethod1` shows the pattern for a test that ensures the analyzer doesn't report a diagnostic when it shouldn't. `TestMethod2` shows the pattern for reporting a diagnostic and running the code fix.

The template uses [Microsoft.CodeAnalysis.Testing](#) packages for unit testing.

TIP

The testing library supports a special markup syntax, including the following:

- `[|text|]` : indicates that a diagnostic is reported for `text`. By default, this form may only be used for testing analyzers with exactly one `DiagnosticDescriptor` provided by `DiagnosticAnalyzer.SupportedDiagnostics`.
- `{|ExpectedDiagnosticId:text|}` : indicates that a diagnostic with `Id` `ExpectedDiagnosticId` is reported for `text`.

Replace the template tests in the `MakeConstUnitTest` class with the following test method:


```

[TestMethod]
public async Task LocalIntCouldBeConstant_Diagnostic()
{
    await VerifyCS.VerifyCodeFixAsync(@"
using System;

class Program
{
    static void Main()
    {
        [|int i = 0;|]
        Console.WriteLine(i);
    }
}
", @"
using System;

class Program
{
    static void Main()
    {
        const int i = 0;
        Console.WriteLine(i);
    }
}
");
}

```

Run this test to make sure it passes. In Visual Studio, open the **Test Explorer** by selecting **Test > Windows > Test Explorer**. Then select **Run All**.

Create tests for valid declarations

As a general rule, analyzers should exit as quickly as possible, doing minimal work. Visual Studio calls registered analyzers as the user edits code. Responsiveness is a key requirement. There are several test cases for code that should not raise your diagnostic. Your analyzer already handles one of those tests, the case where a variable is assigned after being initialized. Add the following test method to represent that case:

```

[TestMethod]
public async Task VariableIsAssigned_NoDiagnostic()
{
    await VerifyCS.VerifyAnalyzerAsync(@"
using System;

class Program
{
    static void Main()
    {
        int i = 0;
        Console.WriteLine(i++);
    }
}
");
}

```

This test passes as well. Next, add test methods for conditions you haven't handled yet:

- Declarations that are already `const`, because they are already const:

```

[TestMethod]
public async Task VariableIsAlreadyConst_NoDiagnostic()
{
    await VerifyCS.VerifyAnalyzerAsync(@"
using System;

class Program
{
    static void Main()
    {
        const int i = 0;
        Console.WriteLine(i);
    }
}
");
}

```

- Declarations that have no initializer, because there is no value to use:

```

[TestMethod]
public async Task NoInitializer_NoDiagnostic()
{
    await VerifyCS.VerifyAnalyzerAsync(@"
using System;

class Program
{
    static void Main()
    {
        int i;
        i = 0;
        Console.WriteLine(i);
    }
}
");
}

```

- Declarations where the initializer is not a constant, because they can't be compile-time constants:

```

[TestMethod]
public async Task InitializerIsNotConstant_NoDiagnostic()
{
    await VerifyCS.VerifyAnalyzerAsync(@"
using System;

class Program
{
    static void Main()
    {
        int i = DateTime.Now.DayOfYear;
        Console.WriteLine(i);
    }
}
");
}

```

It can be even more complicated because C# allows multiple declarations as one statement. Consider the following test case string constant:

```

[TestMethod]
public async Task MultipleInitializers_NoDiagnostic()
{
    await VerifyCS.VerifyAnalyzerAsync(@"
using System;

class Program
{
    static void Main()
    {
        int i = 0, j = DateTime.Now.DayOfYear;
        Console.WriteLine(i);
        Console.WriteLine(j);
    }
}
");
}

```

The variable `i` can be made constant, but the variable `j` cannot. Therefore, this statement cannot be made a `const` declaration.

Run your tests again, and you'll see these new test cases fail.

Update your analyzer to ignore correct declarations

You need some enhancements to your analyzer's `AnalyzeNode` method to filter out code that matches these conditions. They are all related conditions, so similar changes will fix all these conditions. Make the following changes to `AnalyzeNode`:

- Your semantic analysis examined a single variable declaration. This code needs to be in a `foreach` loop that examines all the variables declared in the same statement.
- Each declared variable needs to have an initializer.
- Each declared variable's initializer must be a compile-time constant.

In your `AnalyzeNode` method, replace the original semantic analysis:

```

// Perform data flow analysis on the local declaration.
DataFlowAnalysis dataFlowAnalysis = context.SemanticModel.AnalyzeDataFlow(localDeclaration);

// Retrieve the local symbol for each variable in the local declaration
// and ensure that it is not written outside of the data flow analysis region.
VariableDeclaratorSyntax variable = localDeclaration.Declaration.Variables.Single();
ISymbol variableSymbol = context.SemanticModel.GetDeclaredSymbol(variable, context.CancellationToken);
if (dataFlowAnalysis.WrittenOutside.Contains(variableSymbol))
{
    return;
}

```

with the following code snippet:

```

// Ensure that all variables in the local declaration have initializers that
// are assigned with constant values.
foreach (VariableDeclaratorSyntax variable in localDeclaration.Declaration.Variables)
{
    EqualsValueClauseSyntax initializer = variable.Initializer;
    if (initializer == null)
    {
        return;
    }

    Optional<object> constantValue = context.SemanticModel.GetConstantValue(initializer.Value,
context.CancellationToken);
    if (!constantValue.HasValue)
    {
        return;
    }
}

// Perform data flow analysis on the local declaration.
DataFlowAnalysis dataFlowAnalysis = context.SemanticModel.AnalyzeDataFlow(localDeclaration);

foreach (VariableDeclaratorSyntax variable in localDeclaration.Declaration.Variables)
{
    // Retrieve the local symbol for each variable in the local declaration
    // and ensure that it is not written outside of the data flow analysis region.
    ISymbol variableSymbol = context.SemanticModel.GetDeclaredSymbol(variable, context.CancellationToken);
    if (dataFlowAnalysis.WrittenOutside.Contains(variableSymbol))
    {
        return;
    }
}
}

```

The first `foreach` loop examines each variable declaration using syntactic analysis. The first check guarantees that the variable has an initializer. The second check guarantees that the initializer is a constant. The second loop has the original semantic analysis. The semantic checks are in a separate loop because it has a greater impact on performance. Run your tests again, and you should see them all pass.

Add the final polish

You're almost done. There are a few more conditions for your analyzer to handle. Visual Studio calls analyzers while the user is writing code. It's often the case that your analyzer will be called for code that doesn't compile. The diagnostic analyzer's `AnalyzeNode` method does not check to see if the constant value is convertible to the variable type. So, the current implementation will happily convert an incorrect declaration such as

```
int i = "abc"
```

to a local constant. Add a test method for this case:

```

[TestMethod]
public async Task DeclarationIsValid_NoDiagnostic()
{
    await VerifyCS.VerifyAnalyzerAsync(@"
using System;

class Program
{
    static void Main()
    {
        int x = {|CS0029:""abc""|};
    }
}
");
}

```

In addition, reference types are not handled properly. The only constant value allowed for a reference type is `null`, except in the case of `System.String`, which allows string literals. In other words, `const string s = "abc"` is legal, but `const object s = "abc"` is not. This code snippet verifies that condition:

```
[TestMethod]
public async Task DeclarationIsNotString_NoDiagnostic()
{
    await VerifyCS.VerifyAnalyzerAsync(@"
using System;

class Program
{
    static void Main()
    {
        object s = ""abc"";
    }
}
");
}
```

To be thorough, you need to add another test to make sure that you can create a constant declaration for a string. The following snippet defines both the code that raises the diagnostic, and the code after the fix has been applied:

```
[TestMethod]
public async Task StringCouldBeConstant_Diagnostic()
{
    await VerifyCS.VerifyCodeFixAsync(@"
using System;

class Program
{
    static void Main()
    {
        [|string s = ""abc"";|]
    }
}
", @"
using System;

class Program
{
    static void Main()
    {
        const string s = ""abc"";
    }
}
");
}
```

Finally, if a variable is declared with the `var` keyword, the code fix does the wrong thing and generates a `const var` declaration, which is not supported by the C# language. To fix this bug, the code fix must replace the `var` keyword with the inferred type's name:

```

        [TestMethod]
        public async Task VarIntDeclarationCouldBeConstant_Diagnostic()
        {
            await VerifyCS.VerifyCodeFixAsync(@"
using System;

class Program
{
    static void Main()
    {
        [|var item = 4;|]
    }
}
", @"
using System;

class Program
{
    static void Main()
    {
        const int item = 4;
    }
}
");

        [TestMethod]
        public async Task VarStringDeclarationCouldBeConstant_Diagnostic()
        {
            await VerifyCS.VerifyCodeFixAsync(@"
using System;

class Program
{
    static void Main()
    {
        [|var item = ""abc"";|]
    }
}
", @"
using System;

class Program
{
    static void Main()
    {
        const string item = ""abc"";
    }
}
");

```

Fortunately, all of the above bugs can be addressed using the same techniques that you just learned.

To fix the first bug, first open *MakeConstAnalyzer.cs* and locate the foreach loop where each of the local declaration's initializers are checked to ensure that they're assigned with constant values. Immediately *before* the first foreach loop, call `context.SemanticModel.GetTypeInfo()` to retrieve detailed information about the declared type of the local declaration:

```

TypeSyntax variableTypeName = localDeclaration.Declaration.Type;
ITypeSymbol variableType = context.SemanticModel.GetTypeInfo(variableTypeName,
context.CancellationToken).ConvertedType;

```

Then, inside your `foreach` loop, check each initializer to make sure it's convertible to the variable type. Add the following check after ensuring that the initializer is a constant:

```
// Ensure that the initializer value can be converted to the type of the
// local declaration without a user-defined conversion.
Conversion conversion = context.SemanticModel.ClassifyConversion(initializer.Value, variableType);
if (!conversion.Exists || conversion.IsUserDefined)
{
    return;
}
```

The next change builds upon the last one. Before the closing curly brace of the first `foreach` loop, add the following code to check the type of the local declaration when the constant is a string or null.

```
// Special cases:
// * If the constant value is a string, the type of the local declaration
//   must be System.String.
// * If the constant value is null, the type of the local declaration must
//   be a reference type.
if (constantValue.Value is string)
{
    if (variableType.SpecialType != SpecialType.System_String)
    {
        return;
    }
}
else if (variableType.IsReferenceType && constantValue.Value != null)
{
    return;
}
```

You must write a bit more code in your code fix provider to replace the `var` keyword with the correct type name. Return to *MakeConstCodeFixProvider.cs*. The code you'll add does the following steps:

- Check if the declaration is a `var` declaration, and if it is:
- Create a new type for the inferred type.
- Make sure the type declaration is not an alias. If so, it is legal to declare `const var`.
- Make sure that `var` isn't a type name in this program. (If so, `const var` is legal).
- Simplify the full type name

That sounds like a lot of code. It's not. Replace the line that declares and initializes `newLocal` with the following code. It goes immediately after the initialization of `newModifiers`:

```

// If the type of the declaration is 'var', create a new type name
// for the inferred type.
VariableDeclarationSyntax variableDeclaration = localDeclaration.Declaration;
TypeSyntax variableTypeName = variableDeclaration.Type;
if (variableTypeName.IsVar)
{
    SemanticModel semanticModel = await
document.GetSemanticModelAsync(cancellationToken).ConfigureAwait(false);

    // Special case: Ensure that 'var' isn't actually an alias to another type
    // (e.g. using var = System.String).
    IAliasSymbol aliasInfo = semanticModel.GetAliasInfo(variableTypeName, cancellationToken);
    if (aliasInfo == null)
    {
        // Retrieve the type inferred for var.
        ITypeSymbol type = semanticModel.GetTypeInfo(variableTypeName, cancellationToken).ConvertedType;

        // Special case: Ensure that 'var' isn't actually a type named 'var'.
        if (type.Name != "var")
        {
            // Create a new TypeSyntax for the inferred type. Be careful
            // to keep any leading and trailing trivia from the var keyword.
            TypeSyntax typeName = SyntaxFactory.ParseTypeName(type.ToString())
                .WithLeadingTrivia(variableTypeName.GetLeadingTrivia())
                .WithTrailingTrivia(variableTypeName.GetTrailingTrivia());

            // Add an annotation to simplify the type name.
            TypeSyntax simplifiedTypeName = typeName.AddAdditionalAnnotations(Simplifier.Annotation);

            // Replace the type in the variable declaration.
            variableDeclaration = variableDeclaration.WithType(simplifiedTypeName);
        }
    }
}
// Produce the new local declaration.
LocalDeclarationStatementSyntax newLocal = trimmedLocal.WithModifiers(newModifiers)
    .WithDeclaration(variableDeclaration);

```

You'll need to add one `using` directive to use the [Simplifier](#) type:

```
using Microsoft.CodeAnalysis.Simplification;
```

Run your tests, and they should all pass. Congratulate yourself by running your finished analyzer. Press **Ctrl+F5** to run the analyzer project in a second instance of Visual Studio with the Roslyn Preview extension loaded.

- In the second Visual Studio instance, create a new C# Console Application project and add `int x = "abc";` to the Main method. Thanks to the first bug fix, no warning should be reported for this local variable declaration (though there's a compiler error as expected).
- Next, add `object s = "abc";` to the Main method. Because of the second bug fix, no warning should be reported.
- Finally, add another local variable that uses the `var` keyword. You'll see that a warning is reported and a suggestion appears beneath to the left.
- Move the editor caret over the squiggly underline and press **Ctrl+.** to display the suggested code fix. Upon selecting your code fix, note that the `var` keyword is now handled correctly.

Finally, add the following code:


```
int i = 2;  
int j = 32;  
int k = i + j;
```

After these changes, you get red squiggles only on the first two variables. Add `const` to both `i` and `j`, and you get a new warning on `k` because it can now be `const`.

Congratulations! You've created your first .NET Compiler Platform extension that performs on-the-fly code analysis to detect an issue and provides a quick fix to correct it. Along the way, you've learned many of the code APIs that are part of the .NET Compiler Platform SDK (Roslyn APIs). You can check your work against the [completed sample](#) in our samples GitHub repository.

Other resources

- [Get started with syntax analysis](#)
- [Get started with semantic analysis](#)

C# programming guide

12/28/2021 • 2 minutes to read • [Edit Online](#)

This section provides detailed information on key C# language features and features accessible to C# through .NET.

Most of this section assumes that you already know something about C# and general programming concepts. If you are a complete beginner with programming or with C#, you might want to visit the [Introduction to C# Tutorials](#) or [.NET In-Browser Tutorial](#), where no prior programming knowledge is required.

For information about specific keywords, operators, and preprocessor directives, see [C# Reference](#). For information about the C# Language Specification, see [C# Language Specification](#).

Program sections

[Inside a C# Program](#)

[Main\(\) and Command-Line Arguments](#)

Language Sections

[Statements, Expressions, and Operators](#)

[Types](#)

[Object oriented programming](#)

[Interfaces](#)

[Delegates](#)

[Arrays](#)

[Strings](#)

[Properties](#)

[Indexers](#)

[Events](#)

[Generics](#)

[Iterators](#)

[LINQ Query Expressions](#)

[Namespaces](#)

[Unsafe Code and Pointers](#)

[XML Documentation Comments](#)

Platform Sections

[Application Domains](#)

[Assemblies in .NET](#)

[Attributes](#)

[Collections](#)

[Exceptions and Exception Handling](#)

[File System and the Registry \(C# Programming Guide\)](#)

[Interoperability](#)

[Reflection](#)

See also

- [C# Reference](#)

Programming Concepts (C#)

12/28/2021 • 2 minutes to read • [Edit Online](#)

This section explains programming concepts in the C# language.

In This Section

TITLE	DESCRIPTION
Assemblies in .NET	Describes how to create and use assemblies.
Asynchronous Programming with async and await (C#)	Describes how to write asynchronous solutions by using the async and await keywords in C#. Includes a walkthrough.
Attributes (C#)	Discusses how to provide additional information about programming elements such as types, fields, methods, and properties by using attributes.
Collections (C#)	Describes some of the types of collections provided by .NET. Demonstrates how to use simple collections and collections of key/value pairs.
Covariance and Contravariance (C#)	Shows how to enable implicit conversion of generic type parameters in interfaces and delegates.
Expression Trees (C#)	Explains how you can use expression trees to enable dynamic modification of executable code.
Iterators (C#)	Describes iterators, which are used to step through collections and return elements one at a time.
Language-Integrated Query (LINQ) (C#)	Discusses the powerful query capabilities in the language syntax of C#, and the model for querying relational databases, XML documents, datasets, and in-memory collections.
Reflection (C#)	Explains how to use reflection to dynamically create an instance of a type, bind the type to an existing object, or get the type from an existing object and invoke its methods or access its fields and properties.
Serialization (C#)	Describes key concepts in binary, XML, and SOAP serialization.

Related Sections

- [Performance Tips](#)

Discusses several basic rules that may help you increase the performance of your application.

Asynchronous programming with async and await

12/28/2021 • 15 minutes to read • [Edit Online](#)

The [Task asynchronous programming model \(TAP\)](#) provides an abstraction over asynchronous code. You write code as a sequence of statements, just like always. You can read that code as though each statement completes before the next begins. The compiler performs many transformations because some of those statements may start work and return a [Task](#) that represents the ongoing work.

That's the goal of this syntax: enable code that reads like a sequence of statements, but executes in a much more complicated order based on external resource allocation and when tasks complete. It's analogous to how people give instructions for processes that include asynchronous tasks. Throughout this article, you'll use an example of instructions for making a breakfast to see how the `async` and `await` keywords make it easier to reason about code, that includes a series of asynchronous instructions. You'd write the instructions something like the following list to explain how to make a breakfast:

1. Pour a cup of coffee.
2. Heat up a pan, then fry two eggs.
3. Fry three slices of bacon.
4. Toast two pieces of bread.
5. Add butter and jam to the toast.
6. Pour a glass of orange juice.

If you have experience with cooking, you'd execute those instructions **asynchronously**. You'd start warming the pan for eggs, then start the bacon. You'd put the bread in the toaster, then start the eggs. At each step of the process, you'd start a task, then turn your attention to tasks that are ready for your attention.

Cooking breakfast is a good example of asynchronous work that isn't parallel. One person (or thread) can handle all these tasks. Continuing the breakfast analogy, one person can make breakfast asynchronously by starting the next task before the first completes. The cooking progresses whether or not someone is watching it. As soon as you start warming the pan for the eggs, you can begin frying the bacon. Once the bacon starts, you can put the bread into the toaster.

For a parallel algorithm, you'd need multiple cooks (or threads). One would make the eggs, one the bacon, and so on. Each one would be focused on just that one task. Each cook (or thread) would be blocked synchronously waiting for bacon to be ready to flip, or the toast to pop.

Now, consider those same instructions written as C# statements:

```
using System;
using System.Threading.Tasks;

namespace AsyncBreakfast
{
    class Program
    {
        static void Main(string[] args)
        {
            Coffee cup = PourCoffee();
            Console.WriteLine("coffee is ready");

            Egg eggs = FryEggs(2);
            Console.WriteLine("eggs are ready");

            Bacon bacon = FryBacon(3);
```

```

        Console.WriteLine("bacon is ready");

        Toast toast = ToastBread(2);
        ApplyButter(toast);
        ApplyJam(toast);
        Console.WriteLine("toast is ready");

        Juice oj = PourOJ();
        Console.WriteLine("oj is ready");
        Console.WriteLine("Breakfast is ready!");
    }

    private static Juice PourOJ()
    {
        Console.WriteLine("Pouring orange juice");
        return new Juice();
    }

    private static void ApplyJam(Toast toast) =>
        Console.WriteLine("Putting jam on the toast");

    private static void ApplyButter(Toast toast) =>
        Console.WriteLine("Putting butter on the toast");

    private static Toast ToastBread(int slices)
    {
        for (int slice = 0; slice < slices; slice++)
        {
            Console.WriteLine("Putting a slice of bread in the toaster");
        }
        Console.WriteLine("Start toasting...");
        Task.Delay(3000).Wait();
        Console.WriteLine("Remove toast from toaster");

        return new Toast();
    }

    private static Bacon FryBacon(int slices)
    {
        Console.WriteLine($"putting {slices} slices of bacon in the pan");
        Console.WriteLine("cooking first side of bacon...");
        Task.Delay(3000).Wait();
        for (int slice = 0; slice < slices; slice++)
        {
            Console.WriteLine("flipping a slice of bacon");
        }
        Console.WriteLine("cooking the second side of bacon...");
        Task.Delay(3000).Wait();
        Console.WriteLine("Put bacon on plate");

        return new Bacon();
    }

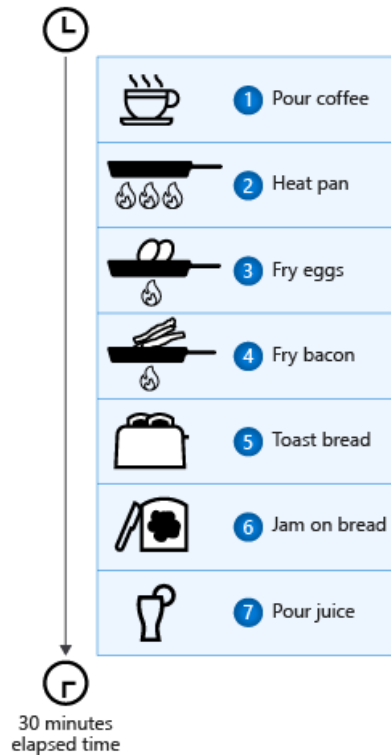
    private static Egg FryEggs(int howMany)
    {
        Console.WriteLine("Warming the egg pan...");
        Task.Delay(3000).Wait();
        Console.WriteLine($"cracking {howMany} eggs");
        Console.WriteLine("cooking the eggs ...");
        Task.Delay(3000).Wait();
        Console.WriteLine("Put eggs on plate");

        return new Egg();
    }

    private static Coffee PourCoffee()
    {
        Console.WriteLine("Pouring coffee");
        return new Coffee();
    }

```

```
}  
}  
}
```



The synchronously prepared breakfast, took roughly 30 minutes because the total is the sum of each individual task.

NOTE

The `Coffee`, `Egg`, `Bacon`, `Toast`, and `Juice` classes are empty. They are simply marker classes for the purpose of demonstration, contain no properties, and serve no other purpose.

Computers don't interpret those instructions the same way people do. The computer will block on each statement until the work is complete before moving on to the next statement. That creates an unsatisfying breakfast. The later tasks wouldn't be started until the earlier tasks had completed. It would take much longer to create the breakfast, and some items would have gotten cold before being served.

If you want the computer to execute the above instructions asynchronously, you must write asynchronous code.

These concerns are important for the programs you write today. When you write client programs, you want the UI to be responsive to user input. Your application shouldn't make a phone appear frozen while it's downloading data from the web. When you write server programs, you don't want threads blocked. Those threads could be serving other requests. Using synchronous code when asynchronous alternatives exist hurts your ability to scale out less expensively. You pay for those blocked threads.

Successful modern applications require asynchronous code. Without language support, writing asynchronous code required callbacks, completion events, or other means that obscured the original intent of the code. The advantage of the synchronous code is that its step-by-step actions make it easy to scan and understand. Traditional asynchronous models forced you to focus on the asynchronous nature of the code, not on the fundamental actions of the code.

Don't block, await instead

The preceding code demonstrates a bad practice: constructing synchronous code to perform asynchronous operations. As written, this code blocks the thread executing it from doing any other work. It won't be interrupted while any of the tasks are in progress. It would be as though you stared at the toaster after putting the bread in. You'd ignore anyone talking to you until the toast popped.

Let's start by updating this code so that the thread doesn't block while tasks are running. The `await` keyword provides a non-blocking way to start a task, then continue execution when that task completes. A simple asynchronous version of the make a breakfast code would look like the following snippet:

```
static async Task Main(string[] args)
{
    Coffee cup = PourCoffee();
    Console.WriteLine("coffee is ready");

    Egg eggs = await FryEggsAsync(2);
    Console.WriteLine("eggs are ready");

    Bacon bacon = await FryBaconAsync(3);
    Console.WriteLine("bacon is ready");

    Toast toast = await ToastBreadAsync(2);
    ApplyButter(toast);
    ApplyJam(toast);
    Console.WriteLine("toast is ready");

    Juice oj = PourOJ();
    Console.WriteLine("oj is ready");
    Console.WriteLine("Breakfast is ready!");
}
```

IMPORTANT

The total elapsed time is roughly the same as the initial synchronous version. The code has yet to take advantage of some of the key features of asynchronous programming.

TIP

The method bodies of the `FryEggsAsync`, `FryBaconAsync`, and `ToastBreadAsync` have all been updated to return `Task<Egg>`, `Task<Bacon>`, and `Task<Toast>` respectively. The methods are renamed from their original version to include the "Async" suffix. Their implementations are shown as part of the [final version](#) later in this article.

This code doesn't block while the eggs or the bacon are cooking. This code won't start any other tasks though. You'd still put the toast in the toaster and stare at it until it pops. But at least, you'd respond to anyone that wanted your attention. In a restaurant where multiple orders are placed, the cook could start another breakfast while the first is cooking.

Now, the thread working on the breakfast isn't blocked while awaiting any started task that hasn't yet finished. For some applications, this change is all that's needed. A GUI application still responds to the user with just this change. However, for this scenario, you want more. You don't want each of the component tasks to be executed sequentially. It's better to start each of the component tasks before awaiting the previous task's completion.

Start tasks concurrently

In many scenarios, you want to start several independent tasks immediately. Then, as each task finishes, you can continue other work that's ready. In the breakfast analogy, that's how you get breakfast done more quickly. You also get everything done close to the same time. You'll get a hot breakfast.

The [System.Threading.Tasks.Task](#) and related types are classes you can use to reason about tasks that are in progress. That enables you to write code that more closely resembles the way you'd actually create breakfast. You'd start cooking the eggs, bacon, and toast at the same time. As each requires action, you'd turn your attention to that task, take care of the next action, then await for something else that requires your attention.

You start a task and hold on to the [Task](#) object that represents the work. You'll `await` each task before working with its result.

Let's make these changes to the breakfast code. The first step is to store the tasks for operations when they start, rather than awaiting them:

```
Coffee cup = PourCoffee();
Console.WriteLine("Coffee is ready");

Task<Egg> eggsTask = FryEggsAsync(2);
Egg eggs = await eggsTask;
Console.WriteLine("Eggs are ready");

Task<Bacon> baconTask = FryBaconAsync(3);
Bacon bacon = await baconTask;
Console.WriteLine("Bacon is ready");

Task<Toast> toastTask = ToastBreadAsync(2);
Toast toast = await toastTask;
ApplyButter(toast);
ApplyJam(toast);
Console.WriteLine("Toast is ready");

Juice oj = PourOJ();
Console.WriteLine("Oj is ready");
Console.WriteLine("Breakfast is ready!");
```

Next, you can move the `await` statements for the bacon and eggs to the end of the method, before serving breakfast:

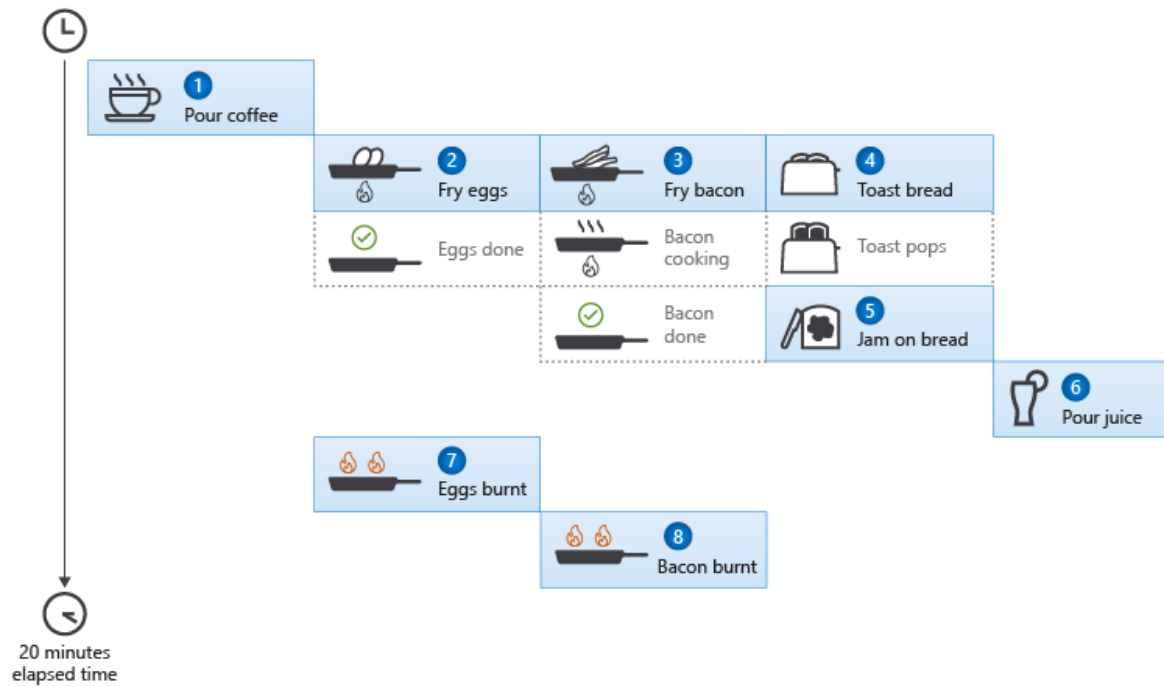
```
Coffee cup = PourCoffee();
Console.WriteLine("Coffee is ready");

Task<Egg> eggsTask = FryEggsAsync(2);
Task<Bacon> baconTask = FryBaconAsync(3);
Task<Toast> toastTask = ToastBreadAsync(2);

Toast toast = await toastTask;
ApplyButter(toast);
ApplyJam(toast);
Console.WriteLine("Toast is ready");
Juice oj = PourOJ();
Console.WriteLine("Oj is ready");

Egg eggs = await eggsTask;
Console.WriteLine("Eggs are ready");
Bacon bacon = await baconTask;
Console.WriteLine("Bacon is ready");

Console.WriteLine("Breakfast is ready!");
```



The asynchronously prepared breakfast took roughly 20 minutes, this time savings is because some tasks ran concurrently.

The preceding code works better. You start all the asynchronous tasks at once. You await each task only when you need the results. The preceding code may be similar to code in a web application that makes requests of different microservices, then combines the results into a single page. You'll make all the requests immediately, then `await` all those tasks and compose the web page.

Composition with tasks

You have everything ready for breakfast at the same time except the toast. Making the toast is the composition of an asynchronous operation (toasting the bread), and synchronous operations (adding the butter and the jam). Updating this code illustrates an important concept:

IMPORTANT

The composition of an asynchronous operation followed by synchronous work is an asynchronous operation. Stated another way, if any portion of an operation is asynchronous, the entire operation is asynchronous.

The preceding code showed you that you can use `Task` or `Task<TResult>` objects to hold running tasks. You `await` each task before using its result. The next step is to create methods that represent the combination of other work. Before serving breakfast, you want to await the task that represents toasting the bread before adding butter and jam. You can represent that work with the following code:

```
static async Task<Toast> MakeToastWithButterAndJamAsync(int number)
{
    var toast = await ToastBreadAsync(number);
    ApplyButter(toast);
    ApplyJam(toast);

    return toast;
}
```

The preceding method has the `async` modifier in its signature. That signals to the compiler that this method contains an `await` statement; it contains asynchronous operations. This method represents the task that toasts

the bread, then adds butter and jam. This method returns a `Task<TResult>` that represents the composition of those three operations. The main block of code now becomes:

```
static async Task Main(string[] args)
{
    Coffee cup = PourCoffee();
    Console.WriteLine("coffee is ready");

    var eggsTask = FryEggsAsync(2);
    var baconTask = FryBaconAsync(3);
    var toastTask = MakeToastWithButterAndJamAsync(2);

    var eggs = await eggsTask;
    Console.WriteLine("eggs are ready");

    var bacon = await baconTask;
    Console.WriteLine("bacon is ready");

    var toast = await toastTask;
    Console.WriteLine("toast is ready");

    Juice oj = PourOJ();
    Console.WriteLine("oj is ready");
    Console.WriteLine("Breakfast is ready!");
}
```

The previous change illustrated an important technique for working with asynchronous code. You compose tasks by separating the operations into a new method that returns a task. You can choose when to await that task. You can start other tasks concurrently.

Asynchronous exceptions

Up to this point, you've implicitly assumed that all these tasks complete successfully. Asynchronous methods throw exceptions, just like their synchronous counterparts. Asynchronous support for exceptions and error handling strives for the same goals as asynchronous support in general: You should write code that reads like a series of synchronous statements. Tasks throw exceptions when they can't complete successfully. The client code can catch those exceptions when a started task is `awaited`. For example, let's assume that the toaster catches fire while making the toast. You can simulate that by modifying the `ToastBreadAsync` method to match the following code:

```
private static async Task<Toast> ToastBreadAsync(int slices)
{
    for (int slice = 0; slice < slices; slice++)
    {
        Console.WriteLine("Putting a slice of bread in the toaster");
    }
    Console.WriteLine("Start toasting...");
    await Task.Delay(2000);
    Console.WriteLine("Fire! Toast is ruined!");
    throw new InvalidOperationException("The toaster is on fire");
    await Task.Delay(1000);
    Console.WriteLine("Remove toast from toaster");

    return new Toast();
}
```

NOTE

You'll get a warning when you compile the preceding code regarding unreachable code. That's intentional, because once the toaster catches fire, operations won't proceed normally.

Run the application after making these changes, and you'll output similar to the following text:

```
Pouring coffee
Coffee is ready
Warming the egg pan...
putting 3 slices of bacon in the pan
Cooking first side of bacon...
Putting a slice of bread in the toaster
Putting a slice of bread in the toaster
Start toasting...
Fire! Toast is ruined!
Flipping a slice of bacon
Flipping a slice of bacon
Flipping a slice of bacon
Cooking the second side of bacon...
Cracking 2 eggs
Cooking the eggs ...
Put bacon on plate
Put eggs on plate
Eggs are ready
Bacon is ready
Unhandled exception. System.InvalidOperationException: The toaster is on fire
  at AsyncBreakfast.Program.ToastBreadAsync(Int32 slices) in Program.cs:line 65
  at AsyncBreakfast.Program.MakeToastWithButterAndJamAsync(Int32 number) in Program.cs:line 36
  at AsyncBreakfast.Program.Main(String[] args) in Program.cs:line 24
  at AsyncBreakfast.Program.<Main>(String[] args)
```

Notice that there's quite a few tasks completing between when the toaster catches fire and the exception is observed. When a task that runs asynchronously throws an exception, that Task is *faulted*. The Task object holds the exception thrown in the [Task.Exception](#) property. Faulted tasks throw an exception when they're awaited.

There are two important mechanisms to understand: how an exception is stored in a faulted task, and how an exception is unpackaged and rethrown when code awaits a faulted task.

When code running asynchronously throws an exception, that exception is stored in the `Task`. The [Task.Exception](#) property is an [System.AggregateException](#) because more than one exception may be thrown during asynchronous work. Any exception thrown is added to the [AggregateException.InnerExceptions](#) collection. If that `Exception` property is null, a new `AggregateException` is created and the thrown exception is the first item in the collection.

The most common scenario for a faulted task is that the `Exception` property contains exactly one exception. When code `awaits` a faulted task, the first exception in the [AggregateException.InnerExceptions](#) collection is rethrown. That's why the output from this example shows an `InvalidOperationException` instead of an `AggregateException`. Extracting the first inner exception makes working with asynchronous methods as similar as possible to working with their synchronous counterparts. You can examine the `Exception` property in your code when your scenario may generate multiple exceptions.

Before going on, comment out these two lines in your `ToastBreadAsync` method. You don't want to start another fire:

```
Console.WriteLine("Fire! Toast is ruined!");
throw new InvalidOperationException("The toaster is on fire");
```

Await tasks efficiently

The series of `await` statements at the end of the preceding code can be improved by using methods of the `Task` class. One of those APIs is `WhenAll`, which returns a `Task` that completes when all the tasks in its argument list have completed, as shown in the following code:

```
await Task.WhenAll(eggsTask, baconTask, toastTask);
Console.WriteLine("Eggs are ready");
Console.WriteLine("Bacon is ready");
Console.WriteLine("Toast is ready");
Console.WriteLine("Breakfast is ready!");
```

Another option is to use `WhenAny`, which returns a `Task<Task>` that completes when any of its arguments completes. You can await the returned task, knowing that it has already finished. The following code shows how you could use `WhenAny` to await the first task to finish and then process its result. After processing the result from the completed task, you remove that completed task from the list of tasks passed to `WhenAny`.

```
var breakfastTasks = new List<Task> { eggsTask, baconTask, toastTask };
while (breakfastTasks.Count > 0)
{
    Task finishedTask = await Task.WhenAny(breakfastTasks);
    if (finishedTask == eggsTask)
    {
        Console.WriteLine("Eggs are ready");
    }
    else if (finishedTask == baconTask)
    {
        Console.WriteLine("Bacon is ready");
    }
    else if (finishedTask == toastTask)
    {
        Console.WriteLine("Toast is ready");
    }
    breakfastTasks.Remove(finishedTask);
}
```

After all those changes, the final version of the code looks like this:

```
using System;
using System.Collections.Generic;
using System.Threading.Tasks;

namespace AsyncBreakfast
{
    class Program
    {
        static async Task Main(string[] args)
        {
            Coffee cup = PourCoffee();
            Console.WriteLine("coffee is ready");

            var eggsTask = FryEggsAsync(2);
            var baconTask = FryBaconAsync(3);
            var toastTask = MakeToastWithButterAndJamAsync(2);

            var breakfastTasks = new List<Task> { eggsTask, baconTask, toastTask };
            while (breakfastTasks.Count > 0)
            {
                Task finishedTask = await Task.WhenAny(breakfastTasks);
                if (finishedTask == eggsTask)
                {
                    Console.WriteLine("eggs are ready");
                }
            }
        }
    }
}
```

```

    }
    else if (finishedTask == baconTask)
    {
        Console.WriteLine("bacon is ready");
    }
    else if (finishedTask == toastTask)
    {
        Console.WriteLine("toast is ready");
    }
    breakfastTasks.Remove(finishedTask);
}

Juice oj = PourOJ();
Console.WriteLine("oj is ready");
Console.WriteLine("Breakfast is ready!");
}

static async Task<Toast> MakeToastWithButterAndJamAsync(int number)
{
    var toast = await ToastBreadAsync(number);
    ApplyButter(toast);
    ApplyJam(toast);

    return toast;
}

private static Juice PourOJ()
{
    Console.WriteLine("Pouring orange juice");
    return new Juice();
}

private static void ApplyJam(Toast toast) =>
    Console.WriteLine("Putting jam on the toast");

private static void ApplyButter(Toast toast) =>
    Console.WriteLine("Putting butter on the toast");

private static async Task<Toast> ToastBreadAsync(int slices)
{
    for (int slice = 0; slice < slices; slice++)
    {
        Console.WriteLine("Putting a slice of bread in the toaster");
    }
    Console.WriteLine("Start toasting...");
    await Task.Delay(3000);
    Console.WriteLine("Remove toast from toaster");

    return new Toast();
}

private static async Task<Bacon> FryBaconAsync(int slices)
{
    Console.WriteLine($"putting {slices} slices of bacon in the pan");
    Console.WriteLine("cooking first side of bacon...");
    await Task.Delay(3000);
    for (int slice = 0; slice < slices; slice++)
    {
        Console.WriteLine("flipping a slice of bacon");
    }
    Console.WriteLine("cooking the second side of bacon...");
    await Task.Delay(3000);
    Console.WriteLine("Put bacon on plate");

    return new Bacon();
}

private static async Task<Egg> FryEggsAsync(int howMany)
{

```

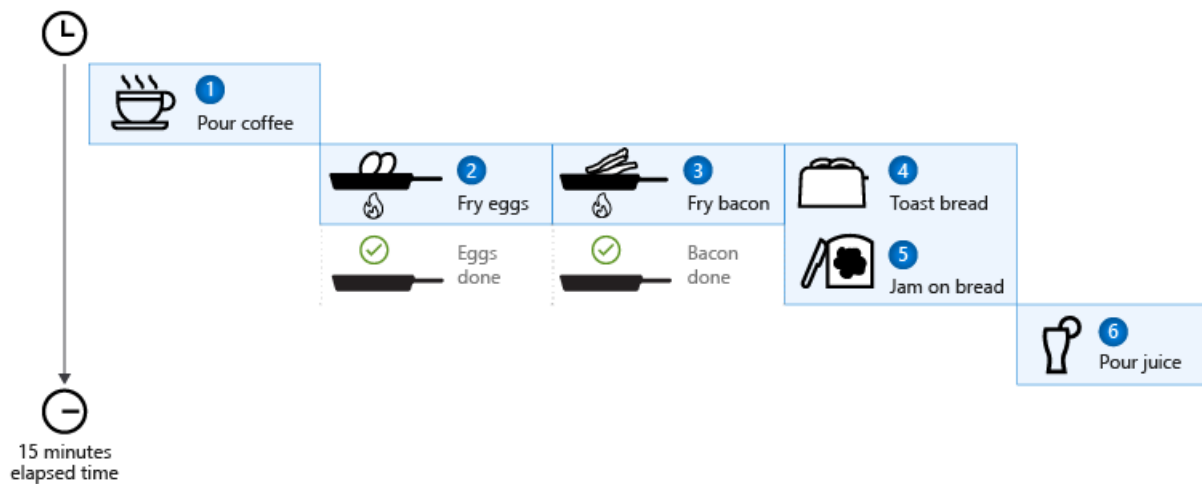
```

        Console.WriteLine("Warming the egg pan...");
        await Task.Delay(3000);
        Console.WriteLine($"cracking {howMany} eggs");
        Console.WriteLine("cooking the eggs ...");
        await Task.Delay(3000);
        Console.WriteLine("Put eggs on plate");

        return new Egg();
    }

    private static Coffee PourCoffee()
    {
        Console.WriteLine("Pouring coffee");
        return new Coffee();
    }
}

```



The final version of the asynchronously prepared breakfast took roughly 15 minutes because some tasks ran concurrently, and the code monitored multiple tasks at once and only took action when it was needed.

This final code is asynchronous. It more accurately reflects how a person would cook a breakfast. Compare the preceding code with the first code sample in this article. The core actions are still clear from reading the code. You can read this code the same way you'd read those instructions for making a breakfast at the beginning of this article. The language features for `async` and `await` provide the translation every person makes to follow those written instructions: start tasks as you can and don't block waiting for tasks to complete.

Next steps

[Explore real world scenarios for asynchronous programs](#)

Asynchronous programming

12/28/2021 • 10 minutes to read • [Edit Online](#)

If you have any I/O-bound needs (such as requesting data from a network, accessing a database, or reading and writing to a file system), you'll want to utilize asynchronous programming. You could also have CPU-bound code, such as performing an expensive calculation, which is also a good scenario for writing async code.

C# has a language-level asynchronous programming model, which allows for easily writing asynchronous code without having to juggle callbacks or conform to a library that supports asynchrony. It follows what is known as the [Task-based Asynchronous Pattern \(TAP\)](#).

Overview of the asynchronous model

The core of async programming is the `Task` and `Task<T>` objects, which model asynchronous operations. They are supported by the `async` and `await` keywords. The model is fairly simple in most cases:

- For I/O-bound code, you await an operation that returns a `Task` or `Task<T>` inside of an `async` method.
- For CPU-bound code, you await an operation that is started on a background thread with the `Task.Run` method.

The `await` keyword is where the magic happens. It yields control to the caller of the method that performed `await`, and it ultimately allows a UI to be responsive or a service to be elastic. While [there are ways](#) to approach async code other than `async` and `await`, this article focuses on the language-level constructs.

I/O-bound example: Download data from a web service

You may need to download some data from a web service when a button is pressed but don't want to block the UI thread. It can be accomplished like this:

```
private readonly HttpClient _httpClient = new HttpClient();

downloadButton.Clicked += async (o, e) =>
{
    // This line will yield control to the UI as the request
    // from the web service is happening.
    //
    // The UI thread is now free to perform other work.
    var stringData = await _httpClient.GetStringAsync(URL);
    DoSomethingWithData(stringData);
};
```

The code expresses the intent (downloading data asynchronously) without getting bogged down in interacting with `Task` objects.

CPU-bound example: Perform a calculation for a game

Say you're writing a mobile game where pressing a button can inflict damage on many enemies on the screen. Performing the damage calculation can be expensive, and doing it on the UI thread would make the game appear to pause as the calculation is performed!

The best way to handle this is to start a background thread, which does the work using `Task.Run`, and await its result using `await`. This allows the UI to feel smooth as the work is being done.


```
private DamageResult CalculateDamageDone()
{
    // Code omitted:
    //
    // Does an expensive calculation and returns
    // the result of that calculation.
}

calculateButton.Clicked += async (o, e) =>
{
    // This line will yield control to the UI while CalculateDamageDone()
    // performs its work. The UI thread is free to perform other work.
    var damageResult = await Task.Run(() => CalculateDamageDone());
    DisplayDamage(damageResult);
};
```

This code clearly expresses the intent of the button's click event, it doesn't require managing a background thread manually, and it does so in a non-blocking way.

What happens under the covers

There are many moving pieces where asynchronous operations are concerned. If you're curious about what's happening underneath the covers of `Task` and `Task<T>`, see the [Async in-depth](#) article for more information.

On the C# side of things, the compiler transforms your code into a state machine that keeps track of things like yielding execution when an `await` is reached and resuming execution when a background job has finished.

For the theoretically inclined, this is an implementation of the [Promise Model of asynchrony](#).

Key pieces to understand

- Async code can be used for both I/O-bound and CPU-bound code, but differently for each scenario.
- Async code uses `Task<T>` and `Task`, which are constructs used to model work being done in the background.
- The `async` keyword turns a method into an async method, which allows you to use the `await` keyword in its body.
- When the `await` keyword is applied, it suspends the calling method and yields control back to its caller until the awaited task is complete.
- `await` can only be used inside an async method.

Recognize CPU-bound and I/O-bound work

The first two examples of this guide showed how you could use `async` and `await` for I/O-bound and CPU-bound work. It's key that you can identify when a job you need to do is I/O-bound or CPU-bound because it can greatly affect the performance of your code and could potentially lead to misusing certain constructs.

Here are two questions you should ask before you write any code:

1. Will your code be "waiting" for something, such as data from a database?

If your answer is "yes", then your work is **I/O-bound**.

2. Will your code be performing an expensive computation?

If you answered "yes", then your work is **CPU-bound**.

If the work you have is **I/O-bound**, use `async` and `await` *without* `Task.Run`. You *should not* use the Task Parallel Library. The reason for this is outlined in [Async in Depth](#).

If the work you have is **CPU-bound** and you care about responsiveness, use `async` and `await`, but spawn off the work on another thread *with* `Task.Run`. If the work is appropriate for concurrency and parallelism, also consider using the [Task Parallel Library](#).

Additionally, you should always measure the execution of your code. For example, you may find yourself in a situation where your CPU-bound work is not costly enough compared with the overhead of context switches when multithreading. Every choice has its tradeoff, and you should pick the correct tradeoff for your situation.

More examples

The following examples demonstrate various ways you can write async code in C#. They cover a few different scenarios you may come across.

Extract data from a network

This snippet downloads the HTML from the homepage at <https://dotnetfoundation.org> and counts the number of times the string ".NET" occurs in the HTML. It uses ASP.NET to define a Web API controller method, which performs this task and returns the number.

NOTE

If you plan on doing HTML parsing in production code, don't use regular expressions. Use a parsing library instead.

```
private readonly HttpClient _httpClient = new HttpClient();

[HttpGet, Route("DotNetCount")]
public async Task<int> GetDotNetCount()
{
    // Suspends GetDotNetCount() to allow the caller (the web server)
    // to accept another request, rather than blocking on this one.
    var html = await _httpClient.GetStringAsync("https://dotnetfoundation.org");

    return Regex.Matches(html, @"\.NET").Count;
}
```

Here's the same scenario written for a Universal Windows App, which performs the same task when a Button is pressed:

```

private readonly HttpClient _httpClient = new HttpClient();

private async void OnSeeTheDotNetsButtonClick(object sender, RoutedEventArgs e)
{
    // Capture the task handle here so we can await the background task later.
    var getDotNetFoundationHtmlTask = _httpClient.GetStringAsync("https://dotnetfoundation.org");

    // Any other work on the UI thread can be done here, such as enabling a Progress Bar.
    // This is important to do here, before the "await" call, so that the user
    // sees the progress bar before execution of this method is yielded.
    NetworkProgressBar.IsEnabled = true;
    NetworkProgressBar.Visibility = Visibility.Visible;

    // The await operator suspends OnSeeTheDotNetsButtonClick(), returning control to its caller.
    // This is what allows the app to be responsive and not block the UI thread.
    var html = await getDotNetFoundationHtmlTask;
    int count = Regex.Matches(html, @"\.NET").Count;

    DotNetCountLabel.Text = $"Number of .NETs on dotnetfoundation.org: {count}";

    NetworkProgressBar.IsEnabled = false;
    NetworkProgressBar.Visibility = Visibility.Collapsed;
}

```

Wait for multiple tasks to complete

You may find yourself in a situation where you need to retrieve multiple pieces of data concurrently. The `Task` API contains two methods, `Task.WhenAll` and `Task.WhenAny`, that allow you to write asynchronous code that performs a non-blocking wait on multiple background jobs.

This example shows how you might grab `User` data for a set of `userId`s.

```

public async Task<User> GetUserAsync(int userId)
{
    // Code omitted:
    //
    // Given a user Id {userId}, retrieves a User object corresponding
    // to the entry in the database with {userId} as its Id.
}

public static async Task<IEnumerable<User>> GetUsersAsync(IEnumerable<int> userIds)
{
    var getUserTasks = new List<Task<User>>();
    foreach (int userId in userIds)
    {
        getUserTasks.Add(GetUserAsync(userId));
    }

    return await Task.WhenAll(getUserTasks);
}

```

Here's another way to write this more succinctly, using LINQ:

```

public async Task<User> GetUserAsync(int userId)
{
    // Code omitted:
    //
    // Given a user Id {userId}, retrieves a User object corresponding
    // to the entry in the database with {userId} as its Id.
}

public static async Task<User[]> GetUsersAsync(IEnumerable<int> userIds)
{
    var getUserTasks = userIds.Select(id => GetUserAsync(id));
    return await Task.WhenAll(getUserTasks);
}

```

Although it's less code, use caution when mixing LINQ with asynchronous code. Because LINQ uses deferred (lazy) execution, async calls won't happen immediately as they do in a `foreach` loop unless you force the generated sequence to iterate with a call to `.ToList()` or `.ToArray()`.

Important info and advice

With async programming, there are some details to keep in mind that can prevent unexpected behavior.

- `async` methods need to have an `await` keyword in their body or they will never yield!

This is important to keep in mind. If `await` is not used in the body of an `async` method, the C# compiler generates a warning, but the code compiles and runs as if it were a normal method. This is incredibly inefficient, as the state machine generated by the C# compiler for the async method is not accomplishing anything.

- Add "Async" as the suffix of every async method name you write.

This is the convention used in .NET to more easily differentiate synchronous and asynchronous methods. Certain methods that aren't explicitly called by your code (such as event handlers or web controller methods) don't necessarily apply. Because they are not explicitly called by your code, being explicit about their naming isn't as important.

- `async void` should only be used for event handlers.

`async void` is the only way to allow asynchronous event handlers to work because events do not have return types (thus cannot make use of `Task` and `Task<T>`). Any other use of `async void` does not follow the TAP model and can be challenging to use, such as:

- Exceptions thrown in an `async void` method can't be caught outside of that method.
- `async void` methods are difficult to test.
- `async void` methods can cause bad side effects if the caller isn't expecting them to be async.

- Tread carefully when using async lambdas in LINQ expressions

Lambda expressions in LINQ use deferred execution, meaning code could end up executing at a time when you're not expecting it to. The introduction of blocking tasks into this can easily result in a deadlock if not written correctly. Additionally, the nesting of asynchronous code like this can also make it more difficult to reason about the execution of the code. Async and LINQ are powerful but should be used together as carefully and clearly as possible.

- Write code that awaits Tasks in a non-blocking manner

Blocking the current thread as a means to wait for a `Task` to complete can result in deadlocks and blocked context threads and can require more complex error-handling. The following table provides guidance on how to deal with waiting for tasks in a non-blocking way:

USE THIS...	INSTEAD OF THIS...	WHEN WISHING TO DO THIS...
<code>await</code>	<code>Task.Wait</code> Or <code>Task.Result</code>	Retrieving the result of a background task
<code>await Task.WhenAny</code>	<code>Task.WaitAny</code>	Waiting for any task to complete
<code>await Task.WhenAll</code>	<code>Task.WaitAll</code>	Waiting for all tasks to complete
<code>await Task.Delay</code>	<code>Thread.Sleep</code>	Waiting for a period of time

- Consider using `ValueTask` where possible

Returning a `Task` object from async methods can introduce performance bottlenecks in certain paths. `Task` is a reference type, so using it means allocating an object. In cases where a method declared with the `async` modifier returns a cached result or completes synchronously, the extra allocations can become a significant time cost in performance critical sections of code. It can become costly if those allocations occur in tight loops. For more information, see [generalized async return types](#).

- Consider using `ConfigureAwait(false)`

A common question is, "when should I use the [Task.ConfigureAwait\(Boolean\)](#) method?". The method allows for a `Task` instance to configure its awaiter. This is an important consideration and setting it incorrectly could potentially have performance implications and even deadlocks. For more information on `ConfigureAwait`, see the [ConfigureAwait FAQ](#).

- Write less stateful code

Don't depend on the state of global objects or the execution of certain methods. Instead, depend only on the return values of methods. Why?

- Code will be easier to reason about.
- Code will be easier to test.
- Mixing async and synchronous code is far simpler.
- Race conditions can typically be avoided altogether.
- Depending on return values makes coordinating async code simple.
- (Bonus) it works really well with dependency injection.

A recommended goal is to achieve complete or near-complete [Referential Transparency](#) in your code. Doing so will result in a predictable, testable, and maintainable codebase.

Other resources

- [Async in-depth](#) provides more information about how Tasks work.
- [The Task asynchronous programming model \(C#\)](#).

Task asynchronous programming model

12/28/2021 • 13 minutes to read • [Edit Online](#)

You can avoid performance bottlenecks and enhance the overall responsiveness of your application by using asynchronous programming. However, traditional techniques for writing asynchronous applications can be complicated, making them difficult to write, debug, and maintain.

[C# 5](#) introduced a simplified approach, async programming, that leverages asynchronous support in the .NET Framework 4.5 and higher, .NET Core, and the Windows Runtime. The compiler does the difficult work that the developer used to do, and your application retains a logical structure that resembles synchronous code. As a result, you get all the advantages of asynchronous programming with a fraction of the effort.

This topic provides an overview of when and how to use async programming and includes links to support topics that contain details and examples.

Async improves responsiveness

Asynchrony is essential for activities that are potentially blocking, such as web access. Access to a web resource sometimes is slow or delayed. If such an activity is blocked in a synchronous process, the entire application must wait. In an asynchronous process, the application can continue with other work that doesn't depend on the web resource until the potentially blocking task finishes.

The following table shows typical areas where asynchronous programming improves responsiveness. The listed APIs from .NET and the Windows Runtime contain methods that support async programming.

APPLICATION AREA	.NET TYPES WITH ASYNC METHODS	WINDOWS RUNTIME TYPES WITH ASYNC METHODS
Web access	HttpClient	Windows.Web.Http.HttpClient SyndicationClient
Working with files	JsonSerializer StreamReader StreamWriter XmlReader XmlWriter	StorageFile
Working with images		MediaCapture BitmapEncoder BitmapDecoder
WCF programming	Synchronous and Asynchronous Operations	

Asynchrony proves especially valuable for applications that access the UI thread because all UI-related activity usually shares one thread. If any process is blocked in a synchronous application, all are blocked. Your application stops responding, and you might conclude that it has failed when instead it's just waiting.

When you use asynchronous methods, the application continues to respond to the UI. You can resize or minimize a window, for example, or you can close the application if you don't want to wait for it to finish.

The async-based approach adds the equivalent of an automatic transmission to the list of options that you can choose from when designing asynchronous operations. That is, you get all the benefits of traditional

asynchronous programming but with much less effort from the developer.

Async methods are easy to write

The `async` and `await` keywords in C# are the heart of async programming. By using those two keywords, you can use resources in .NET Framework, .NET Core, or the Windows Runtime to create an asynchronous method almost as easily as you create a synchronous method. Asynchronous methods that you define by using the `async` keyword are referred to as *async methods*.

The following example shows an async method. Almost everything in the code should look familiar to you.

You can find a complete Windows Presentation Foundation (WPF) example available for download from [Asynchronous programming with async and await in C#](#).

```
public async Task<int> GetUrlContentLengthAsync()
{
    var client = new HttpClient();

    Task<string> getStringTask =
        client.GetStringAsync("https://docs.microsoft.com/dotnet");

    DoIndependentWork();

    string contents = await getStringTask;

    return contents.Length;
}

void DoIndependentWork()
{
    Console.WriteLine("Working...");
}
```

You can learn several practices from the preceding sample. Start with the method signature. It includes the `async` modifier. The return type is `Task<int>` (See "Return Types" section for more options). The method name ends in `Async`. In the body of the method, `GetStringAsync` returns a `Task<string>`. That means that when you `await` the task you'll get a `string` (`contents`). Before awaiting the task, you can do work that doesn't rely on the `string` from `GetStringAsync`.

Pay close attention to the `await` operator. It suspends `GetUrlContentLengthAsync`:

- `GetUrlContentLengthAsync` can't continue until `getStringTask` is complete.
- Meanwhile, control returns to the caller of `GetUrlContentLengthAsync`.
- Control resumes here when `getStringTask` is complete.
- The `await` operator then retrieves the `string` result from `getStringTask`.

The return statement specifies an integer result. Any methods that are awaiting `GetUrlContentLengthAsync` retrieve the length value.

If `GetUrlContentLengthAsync` doesn't have any work that it can do between calling `GetStringAsync` and awaiting its completion, you can simplify your code by calling and awaiting in the following single statement.

```
string contents = await client.GetStringAsync("https://docs.microsoft.com/dotnet");
```

The following characteristics summarize what makes the previous example an async method:

- The method signature includes an `async` modifier.

- The name of an async method, by convention, ends with an "Async" suffix.
- The return type is one of the following types:
 - `Task<TResult>` if your method has a return statement in which the operand has type `TResult`.
 - `Task` if your method has no return statement or has a return statement with no operand.
 - `void` if you're writing an async event handler.
 - Any other type that has a `GetAwaiter` method (starting with C# 7.0).

For more information, see the [Return types and parameters](#) section.

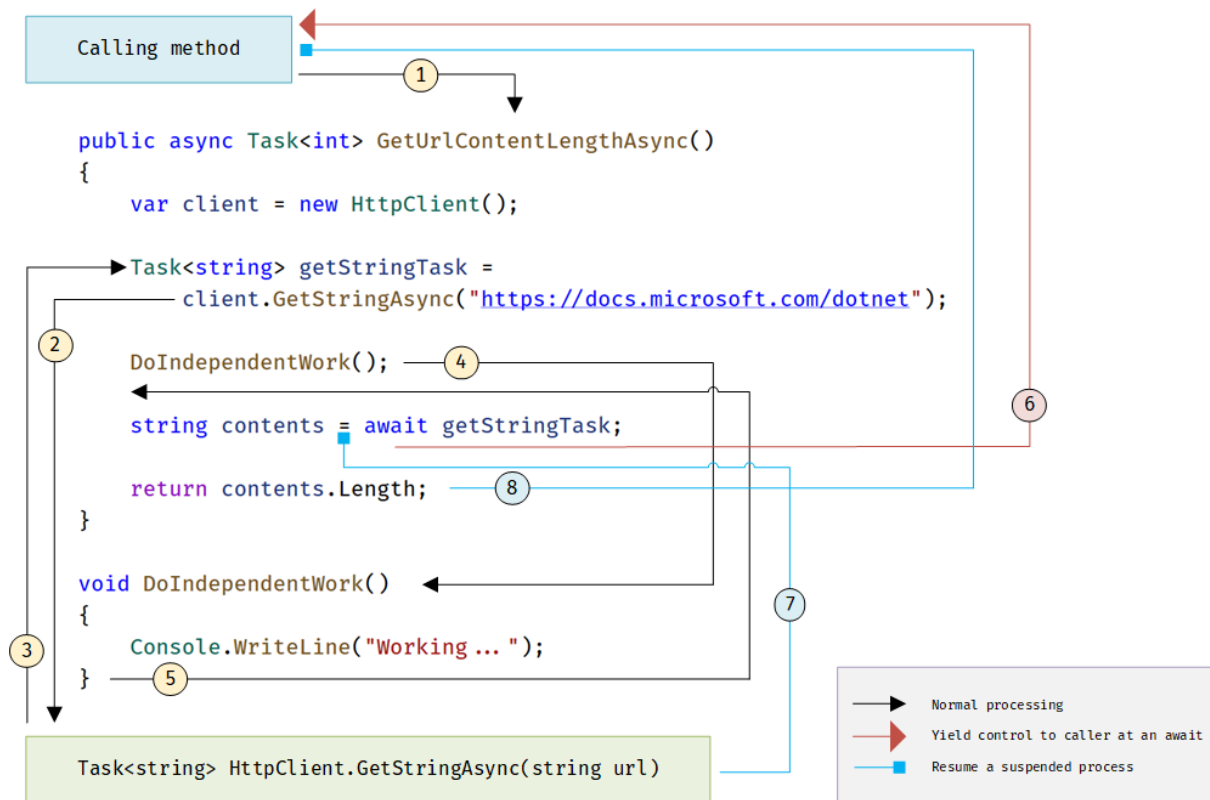
- The method usually includes at least one `await` expression, which marks a point where the method can't continue until the awaited asynchronous operation is complete. In the meantime, the method is suspended, and control returns to the method's caller. The next section of this topic illustrates what happens at the suspension point.

In async methods, you use the provided keywords and types to indicate what you want to do, and the compiler does the rest, including keeping track of what must happen when control returns to an await point in a suspended method. Some routine processes, such as loops and exception handling, can be difficult to handle in traditional asynchronous code. In an async method, you write these elements much as you would in a synchronous solution, and the problem is solved.

For more information about asynchrony in previous versions of .NET Framework, see [TPL and traditional .NET Framework asynchronous programming](#).

What happens in an async method

The most important thing to understand in asynchronous programming is how the control flow moves from method to method. The following diagram leads you through the process:



The numbers in the diagram correspond to the following steps, initiated when a calling method calls the async method.

1. A calling method calls and awaits the `GetUrlContentLengthAsync` async method.

2. `GetUrlContentLengthAsync` creates an `HttpClient` instance and calls the `GetStringAsync` asynchronous method to download the contents of a website as a string.
3. Something happens in `GetStringAsync` that suspends its progress. Perhaps it must wait for a website to download or some other blocking activity. To avoid blocking resources, `GetStringAsync` yields control to its caller, `GetUrlContentLengthAsync`.
- `GetStringAsync` returns a `Task<TResult>`, where `TResult` is a string, and `GetUrlContentLengthAsync` assigns the task to the `getStringTask` variable. The task represents the ongoing process for the call to `GetStringAsync`, with a commitment to produce an actual string value when the work is complete.
4. Because `getStringTask` hasn't been awaited yet, `GetUrlContentLengthAsync` can continue with other work that doesn't depend on the final result from `GetStringAsync`. That work is represented by a call to the synchronous method `DoIndependentWork`.
5. `DoIndependentWork` is a synchronous method that does its work and returns to its caller.
6. `GetUrlContentLengthAsync` has run out of work that it can do without a result from `getStringTask`. `GetUrlContentLengthAsync` next wants to calculate and return the length of the downloaded string, but the method can't calculate that value until the method has the string.

Therefore, `GetUrlContentLengthAsync` uses an await operator to suspend its progress and to yield control to the method that called `GetUrlContentLengthAsync`. `GetUrlContentLengthAsync` returns a `Task<int>` to the caller. The task represents a promise to produce an integer result that's the length of the downloaded string.

NOTE

If `GetStringAsync` (and therefore `getStringTask`) completes before `GetUrlContentLengthAsync` awaits it, control remains in `GetUrlContentLengthAsync`. The expense of suspending and then returning to `GetUrlContentLengthAsync` would be wasted if the called asynchronous process `getStringTask` has already completed and `GetUrlContentLengthAsync` doesn't have to wait for the final result.

Inside the calling method the processing pattern continues. The caller might do other work that doesn't depend on the result from `GetUrlContentLengthAsync` before awaiting that result, or the caller might await immediately. The calling method is waiting for `GetUrlContentLengthAsync`, and `GetUrlContentLengthAsync` is waiting for `GetStringAsync`.

7. `GetStringAsync` completes and produces a string result. The string result isn't returned by the call to `GetStringAsync` in the way that you might expect. (Remember that the method already returned a task in step 3.) Instead, the string result is stored in the task that represents the completion of the method, `getStringTask`. The await operator retrieves the result from `getStringTask`. The assignment statement assigns the retrieved result to `contents`.
8. When `GetUrlContentLengthAsync` has the string result, the method can calculate the length of the string. Then the work of `GetUrlContentLengthAsync` is also complete, and the waiting event handler can resume. In the full example at the end of the topic, you can confirm that the event handler retrieves and prints the value of the length result. If you are new to asynchronous programming, take a minute to consider the difference between synchronous and asynchronous behavior. A synchronous method returns when its work is complete (step 5), but an async method returns a task value when its work is suspended (steps 3 and 6). When the async method eventually completes its work, the task is marked as completed and the result, if any, is stored in the task.

API async methods

You might be wondering where to find methods such as `GetStringAsync` that support async programming. .NET Framework 4.5 or higher and .NET Core contain many members that work with `async` and `await`. You can recognize them by the "Async" suffix that's appended to the member name, and by their return type of `Task` or `Task<TResult>`. For example, the `System.IO.Stream` class contains methods such as `CopyToAsync`, `ReadAsync`, and `WriteAsync` alongside the synchronous methods `CopyTo`, `Read`, and `Write`.

The Windows Runtime also contains many methods that you can use with `async` and `await` in Windows apps. For more information, see [Threading and async programming](#) for UWP development, and [Asynchronous programming \(Windows Store apps\)](#) and [Quickstart: Calling asynchronous APIs in C# or Visual Basic](#) if you use earlier versions of the Windows Runtime.

Threads

Async methods are intended to be non-blocking operations. An `await` expression in an async method doesn't block the current thread while the awaited task is running. Instead, the expression signs up the rest of the method as a continuation and returns control to the caller of the async method.

The `async` and `await` keywords don't cause additional threads to be created. Async methods don't require multithreading because an async method doesn't run on its own thread. The method runs on the current synchronization context and uses time on the thread only when the method is active. You can use `Task.Run` to move CPU-bound work to a background thread, but a background thread doesn't help with a process that's just waiting for results to become available.

The async-based approach to asynchronous programming is preferable to existing approaches in almost every case. In particular, this approach is better than the `BackgroundWorker` class for I/O-bound operations because the code is simpler and you don't have to guard against race conditions. In combination with the `Task.Run` method, async programming is better than `BackgroundWorker` for CPU-bound operations because async programming separates the coordination details of running your code from the work that `Task.Run` transfers to the thread pool.

async and await

If you specify that a method is an async method by using the `async` modifier, you enable the following two capabilities.

- The marked async method can use `await` to designate suspension points. The `await` operator tells the compiler that the async method can't continue past that point until the awaited asynchronous process is complete. In the meantime, control returns to the caller of the async method.

The suspension of an async method at an `await` expression doesn't constitute an exit from the method, and `finally` blocks don't run.

- The marked async method can itself be awaited by methods that call it.

An async method typically contains one or more occurrences of an `await` operator, but the absence of `await` expressions doesn't cause a compiler error. If an async method doesn't use an `await` operator to mark a suspension point, the method executes as a synchronous method does, despite the `async` modifier. The compiler issues a warning for such methods.

`async` and `await` are contextual keywords. For more information and examples, see the following topics:

- [async](#)
- [await](#)

Return types and parameters

An async method typically returns a [Task](#) or a [Task<TResult>](#). Inside an async method, an `await` operator is applied to a task that's returned from a call to another async method.

You specify [Task<TResult>](#) as the return type if the method contains a `return` statement that specifies an operand of type `TResult`.

You use [Task](#) as the return type if the method has no return statement or has a return statement that doesn't return an operand.

Starting with C# 7.0, you can also specify any other return type, provided that the type includes a `GetAwaiter` method. [ValueTask<TResult>](#) is an example of such a type. It is available in the [System.Threading.Tasks.Extensions](#) NuGet package.

The following example shows how you declare and call a method that returns a [Task<TResult>](#) or a [Task](#):

```
async Task<int> GetTaskOfTResultAsync()
{
    int hours = 0;
    await Task.Delay(0);

    return hours;
}

Task<int> returnedTaskTResult = GetTaskOfTResultAsync();
int intResult = await returnedTaskTResult;
// Single line
// int intResult = await GetTaskOfTResultAsync();

async Task GetTaskAsync()
{
    await Task.Delay(0);
    // No return statement needed
}

Task returnedTask = GetTaskAsync();
await returnedTask;
// Single line
await GetTaskAsync();
```

Each returned task represents ongoing work. A task encapsulates information about the state of the asynchronous process and, eventually, either the final result from the process or the exception that the process raises if it doesn't succeed.

An async method can also have a `void` return type. This return type is used primarily to define event handlers, where a `void` return type is required. Async event handlers often serve as the starting point for async programs.

An async method that has a `void` return type can't be awaited, and the caller of a void-returning method can't catch any exceptions that the method throws.

An async method can't declare [in](#), [ref](#) or [out](#) parameters, but the method can call methods that have such parameters. Similarly, an async method can't return a value by reference, although it can call methods with [ref](#) return values.

For more information and examples, see [Async return types \(C#\)](#). For more information about how to catch exceptions in async methods, see [try-catch](#).

Asynchronous APIs in Windows Runtime programming have one of the following return types, which are similar to tasks:

- [IAsyncOperation<TResult>](#), which corresponds to [Task<TResult>](#)
- [IAsyncAction](#), which corresponds to [Task](#)
- [IAsyncActionWithProgress<TProgress>](#)
- [IAsyncOperationWithProgress<TResult,TProgress>](#)

Naming convention

By convention, methods that return commonly awaitable types (for example, `Task`, `Task<T>`, `ValueTask`, `ValueTask<T>`) should have names that end with "Async". Methods that start an asynchronous operation but do not return an awaitable type should not have names that end with "Async", but may start with "Begin", "Start", or some other verb to suggest this method does not return or throw the result of the operation.

You can ignore the convention where an event, base class, or interface contract suggests a different name. For example, you shouldn't rename common event handlers, such as `OnButtonClick`.

Related articles (Visual Studio)

TITLE	DESCRIPTION
How to make multiple web requests in parallel by using async and await (C#)	Demonstrates how to start several tasks at the same time.
Async return types (C#)	Illustrates the types that async methods can return, and explains when each type is appropriate.
Cancel tasks with a cancellation token as a signaling mechanism.	Shows how to add the following functionality to your async solution: <ul style="list-style-type: none"> - Cancel a list of tasks (C#) - Cancel tasks after a period of time (C#) - Process asynchronous task as they complete (C#)
Using async for file access (C#)	Lists and demonstrates the benefits of using async and await to access files.
Task-based asynchronous pattern (TAP)	Describes an asynchronous pattern, the pattern is based on the Task and Task<TResult> types.
Async Videos on Channel 9	Provides links to a variety of videos about async programming.

See also

- [async](#)
- [await](#)
- [Asynchronous programming](#)
- [Async overview](#)

Async return types (C#)

12/28/2021 • 9 minutes to read • [Edit Online](#)

Async methods can have the following return types:

- [Task](#), for an async method that performs an operation but returns no value.
- [Task<TResult>](#), for an async method that returns a value.
- `void`, for an event handler.
- Starting with C# 7.0, any type that has an accessible `GetAwaiter` method. The object returned by the `GetAwaiter` method must implement the [System.Runtime.CompilerServices.ICriticalNotifyCompletion](#) interface.
- Starting with C# 8.0, [IAsyncEnumerable<T>](#), for an async method that returns an *async stream*.

For more information about async methods, see [Asynchronous programming with async and await \(C#\)](#).

Several other types also exist that are specific to Windows workloads:

- [DispatcherOperation](#), for async operations limited to Windows.
- [IAsyncAction](#), for async actions in UWP that don't return a value.
- [IAsyncActionWithProgress<TProgress>](#), for async actions in UWP that report progress but don't return a value.
- [IAsyncOperation<TResult>](#), for async operations in UWP that return a value.
- [IAsyncOperationWithProgress<TResult,TProgress>](#), for async operations in UWP that report progress and return a value.

Task return type

Async methods that don't contain a `return` statement or that contain a `return` statement that doesn't return an operand usually have a return type of [Task](#). Such methods return `void` if they run synchronously. If you use a [Task](#) return type for an async method, a calling method can use an `await` operator to suspend the caller's completion until the called async method has finished.

In the following example, the `WaitAndApologizeAsync` method doesn't contain a `return` statement, so the method returns a [Task](#) object. Returning a `Task` enables `WaitAndApologizeAsync` to be awaited. The [Task](#) type doesn't include a `Result` property because it has no return value.

```

public static async Task DisplayCurrentInfoAsync()
{
    await WaitAndApologizeAsync();

    Console.WriteLine($"Today is {DateTime.Now:D}");
    Console.WriteLine($"The current time is {DateTime.Now.TimeOfDay:t}");
    Console.WriteLine("The current temperature is 76 degrees.");
}

static async Task WaitAndApologizeAsync()
{
    await Task.Delay(2000);

    Console.WriteLine("Sorry for the delay...\n");
}
// Example output:
//     Sorry for the delay...
//
// Today is Monday, August 17, 2020
// The current time is 12:59:24.2183304
// The current temperature is 76 degrees.

```

`WaitAndApologizeAsync` is awaited by using an `await` statement instead of an `await` expression, similar to the calling statement for a synchronous void-returning method. The application of an `await` operator in this case doesn't produce a value. When the right operand of an `await` is a `Task<TResult>`, the `await` expression produces a result of `T`. When the right operand of an `await` is a `Task`, the `await` and its operand are a statement.

You can separate the call to `WaitAndApologizeAsync` from the application of an `await` operator, as the following code shows. However, remember that a `Task` doesn't have a `Result` property, and that no value is produced when an `await` operator is applied to a `Task`.

The following code separates calling the `WaitAndApologizeAsync` method from awaiting the task that the method returns.

```

Task waitAndApologizeTask = WaitAndApologizeAsync();

string output =
    $"Today is {DateTime.Now:D}\n" +
    $"The current time is {DateTime.Now.TimeOfDay:t}\n" +
    "The current temperature is 76 degrees.\n";

await waitAndApologizeTask;
Console.WriteLine(output);

```

Task<TResult> return type

The `Task<TResult>` return type is used for an async method that contains a `return` statement in which the operand is `TResult`.

In the following example, the `GetLeisureHoursAsync` method contains a `return` statement that returns an integer. The method declaration must specify a return type of `Task<int>`. The `FromResult` async method is a placeholder for an operation that returns a `DayOfWeek`.

```

public static async Task ShowTodayInfoAsync()
{
    string message =
        $"Today is {DateTime.Today:D}\n" +
        "Today's hours of leisure: " +
        $"{await GetLeisureHoursAsync()}";

    Console.WriteLine(message);
}

static async Task<int> GetLeisureHoursAsync()
{
    DayOfWeek today = await Task.FromResult(DateTime.Now.DayOfWeek);

    int leisureHours =
        today is DayOfWeek.Saturday || today is DayOfWeek.Sunday
        ? 16 : 5;

    return leisureHours;
}
// Example output:
// Today is Wednesday, May 24, 2017
// Today's hours of leisure: 5

```

When `GetLeisureHoursAsync` is called from within an `await` expression in the `ShowTodayInfo` method, the `await` expression retrieves the integer value (the value of `leisureHours`) that's stored in the task returned by the `GetLeisureHours` method. For more information about `await` expressions, see [await](#).

You can better understand how `await` retrieves the result from a `Task<T>` by separating the call to `GetLeisureHoursAsync` from the application of `await`, as the following code shows. A call to method `GetLeisureHoursAsync` that isn't immediately awaited returns a `Task<int>`, as you would expect from the declaration of the method. The task is assigned to the `getLeisureHoursTask` variable in the example. Because `getLeisureHoursTask` is a `Task<TResult>`, it contains a `Result` property of type `TResult`. In this case, `TResult` represents an integer type. When `await` is applied to `getLeisureHoursTask`, the `await` expression evaluates to the contents of the `Result` property of `getLeisureHoursTask`. The value is assigned to the `ret` variable.

IMPORTANT

The `Result` property is a blocking property. If you try to access it before its task is finished, the thread that's currently active is blocked until the task completes and the value is available. In most cases, you should access the value by using `await` instead of accessing the property directly.

The previous example retrieved the value of the `Result` property to block the main thread so that the `Main` method could print the `message` to the console before the application ended.

```

var getLeisureHoursTask = GetLeisureHoursAsync();

string message =
    $"Today is {DateTime.Today:D}\n" +
    "Today's hours of leisure: " +
    $"{await getLeisureHoursTask}";

Console.WriteLine(message);

```

Void return type

You use the `void` return type in asynchronous event handlers, which require a `void` return type. For methods

other than event handlers that don't return a value, you should return a [Task](#) instead, because an async method that returns `void` can't be awaited. Any caller of such a method must continue to completion without waiting for the called async method to finish. The caller must be independent of any values or exceptions that the async method generates.

The caller of a void-returning async method can't catch exceptions thrown from the method. Such unhandled exceptions are likely to cause your application to fail. If a method that returns a [Task](#) or [Task<TResult>](#) throws an exception, the exception is stored in the returned task. The exception is rethrown when the task is awaited. Make sure that any async method that can produce an exception has a return type of [Task](#) or [Task<TResult>](#) and that calls to the method are awaited.

For more information about how to catch exceptions in async methods, see the [Exceptions in async methods](#) section of the [try-catch](#) article.

The following example shows the behavior of an async event handler. In the example code, an async event handler must let the main thread know when it finishes. Then the main thread can wait for an async event handler to complete before exiting the program.

```
using System;
using System.Threading.Tasks;

public class NaiveButton
{
    public event EventHandler? Clicked;

    public void Click()
    {
        Console.WriteLine("Somebody has clicked a button. Let's raise the event...");
        Clicked?.Invoke(this, EventArgs.Empty);
        Console.WriteLine("All listeners are notified.");
    }
}

public class AsyncVoidExample
{
    static readonly TaskCompletionSource<bool> s_tcs = new TaskCompletionSource<bool>();

    public static async Task MultipleEventHandlersAsync()
    {
        Task<bool> secondHandlerFinished = s_tcs.Task;

        var button = new NaiveButton();

        button.Clicked += OnButtonClicked1;
        button.Clicked += OnButtonClicked2Async;
        button.Clicked += OnButtonClicked3;

        Console.WriteLine("Before button.Click() is called...");
        button.Click();
        Console.WriteLine("After button.Click() is called...");

        await secondHandlerFinished;
    }

    private static void OnButtonClicked1(object? sender, EventArgs e)
    {
        Console.WriteLine("  Handler 1 is starting...");
        Task.Delay(100).Wait();
        Console.WriteLine("  Handler 1 is done.");
    }

    private static async void OnButtonClicked2Async(object? sender, EventArgs e)
    {
        Console.WriteLine("  Handler 2 is starting...");
        Task.Delay(100).Wait();
    }

    private static void OnButtonClicked3(object? sender, EventArgs e)
    {
        Console.WriteLine("  Handler 3 is starting...");
        Task.Delay(100).Wait();
    }
}
```



```

        task.Delay(100).Wait();
        Console.WriteLine("    Handler 2 is about to go async...");
        await Task.Delay(500);
        Console.WriteLine("    Handler 2 is done.");
        s_tcs.SetResult(true);
    }

    private static void OnButtonClicked3(object? sender, EventArgs e)
    {
        Console.WriteLine("    Handler 3 is starting...");
        Task.Delay(100).Wait();
        Console.WriteLine("    Handler 3 is done.");
    }
}

// Example output:
//
// Before button.Click() is called...
// Somebody has clicked a button. Let's raise the event...
//    Handler 1 is starting...
//    Handler 1 is done.
//    Handler 2 is starting...
//    Handler 2 is about to go async...
//    Handler 3 is starting...
//    Handler 3 is done.
// All listeners are notified.
// After button.Click() is called...
//    Handler 2 is done.

```

Generalized async return types and ValueTask<TResult>

Starting with C# 7.0, an async method can return any type that has an accessible `GetAwaiter` method that returns an instance of an *awaiter type*. In addition, the type returned from the `GetAwaiter` method must have the [System.Runtime.CompilerServices.AsyncMethodBuilderAttribute](#) attribute. You can learn more in the article on [Attributes read by the compiler](#) or the feature spec for [Task like return types](#).

This feature is the complement to [awaitable expressions](#), which describes the requirements for the operand of `await`. Generalized async return types enable the compiler to generate `async` methods that return different types. Generalized async return types enabled performance improvements in the .NET libraries. Because [Task](#) and [Task<TResult>](#) are reference types, memory allocation in performance-critical paths, particularly when allocations occur in tight loops, can adversely affect performance. Support for generalized return types means that you can return a lightweight value type instead of a reference type to avoid additional memory allocations.

.NET provides the [System.Threading.Tasks.ValueTask<TResult>](#) structure as a lightweight implementation of a generalized task-returning value. To use the [System.Threading.Tasks.ValueTask<TResult>](#) type, you must add the `System.Threading.Tasks.Extensions` NuGet package to your project. The following example uses the [ValueTask<TResult>](#) structure to retrieve the value of two dice rolls.

```

using System;
using System.Threading.Tasks;

class Program
{
    static readonly Random s_rnd = new Random();

    static async Task Main() =>
        Console.WriteLine($"You rolled {await GetDiceRollAsync()}");

    static async ValueTask<int> GetDiceRollAsync()
    {
        Console.WriteLine("Shaking dice...");

        int roll1 = await RollAsync();
        int roll2 = await RollAsync();

        return roll1 + roll2;
    }

    static async ValueTask<int> RollAsync()
    {
        await Task.Delay(500);

        int diceRoll = s_rnd.Next(1, 7);
        return diceRoll;
    }
}
// Example output:
//   Shaking dice...
//   You rolled 8

```

Writing a generalized async return type is an advanced scenario, and is targeted for use in specialized environments. Consider using the `Task`, `Task<T>`, and `ValueTask<T>` types instead, which cover most scenarios for asynchronous code.

In C# 10 and later, you can apply the `AsyncMethodBuilder` attribute to an async method (instead of the async return type declaration) to override the builder for that type. Typically you'd apply this attribute to use a different builder provided in the .NET runtime.

Async streams with `IAsyncEnumerable<T>`

Starting with C# 8.0, an async method may return an *async stream*, represented by `IAsyncEnumerable<T>`. An async stream provides a way to enumerate items read from a stream when elements are generated in chunks with repeated asynchronous calls. The following example shows an async method that generates an async stream:

```

static async IEnumerable<string> ReadWordsFromStreamAsync()
{
    string data =
        @"This is a line of text.
        Here is the second line of text.
        And there is one more for good measure.
        Wait, that was the penultimate line.";

    using var readStream = new StringReader(data);

    string line = await readStream.ReadLineAsync();
    while (line != null)
    {
        foreach (string word in line.Split(' ', StringSplitOptions.RemoveEmptyEntries))
        {
            yield return word;
        }

        line = await readStream.ReadLineAsync();
    }
}

```

The preceding example reads lines from a string asynchronously. Once each line is read, the code enumerates each word in the string. Callers would enumerate each word using the `await foreach` statement. The method awaits when it needs to asynchronously read the next line from the source string.

See also

- [FromResult](#)
- [Process asynchronous tasks as they complete](#)
- [Asynchronous programming with async and await \(C#\)](#)
- [async](#)
- [await](#)

Cancel a list of tasks (C#)

12/28/2021 • 4 minutes to read • [Edit Online](#)

You can cancel an async console application if you don't want to wait for it to finish. By following the example in this topic, you can add a cancellation to an application that downloads the contents of a list of websites. You can cancel many tasks by associating the [CancellationTokenSource](#) instance with each task. If you select the Enter key, you cancel all tasks that aren't yet complete.

This tutorial covers:

- Creating a .NET console application
- Writing an async application that supports cancellation
- Demonstrating signaling cancellation

Prerequisites

This tutorial requires the following:

- [.NET 5 or later SDK](#)
- Integrated development environment (IDE)
 - [We recommend Visual Studio, Visual Studio Code, or Visual Studio for Mac](#)

Create example application

Create a new .NET Core console application. You can create one by using the `dotnet new console` command or from [Visual Studio](#). Open the *Program.cs* file in your favorite code editor.

Replace using statements

Replace the existing using statements with these declarations:

```
using System;
using System.Collections.Generic;
using System.Diagnostics;
using System.Net.Http;
using System.Threading;
using System.Threading.Tasks;
```

Add fields

In the `Program` class definition, add these three fields:

```

static readonly CancellationTokenSource s_cts = new CancellationTokenSource();

static readonly HttpClient s_client = new HttpClient
{
    MaxResponseContentBufferSize = 1_000_000
};

static readonly IEnumerable<string> s_urllist = new string[]
{
    "https://docs.microsoft.com",
    "https://docs.microsoft.com/aspnet/core",
    "https://docs.microsoft.com/azure",
    "https://docs.microsoft.com/azure/devops",
    "https://docs.microsoft.com/dotnet",
    "https://docs.microsoft.com/dynamics365",
    "https://docs.microsoft.com/education",
    "https://docs.microsoft.com/enterprise-mobility-security",
    "https://docs.microsoft.com/gaming",
    "https://docs.microsoft.com/graph",
    "https://docs.microsoft.com/microsoft-365",
    "https://docs.microsoft.com/office",
    "https://docs.microsoft.com/powershell",
    "https://docs.microsoft.com/sql",
    "https://docs.microsoft.com/surface",
    "https://docs.microsoft.com/system-center",
    "https://docs.microsoft.com/visualstudio",
    "https://docs.microsoft.com/windows",
    "https://docs.microsoft.com/xamarin"
};

```

The [CancellationTokenSource](#) is used to signal a requested cancellation to a [CancellationToken](#). The `HttpClient` exposes the ability to send HTTP requests and receive HTTP responses. The `s_urllist` holds all of the URLs that the application plans to process.

Update application entry point

The main entry point into the console application is the `Main` method. Replace the existing method with the following:

```

static async Task Main()
{
    Console.WriteLine("Application started.");
    Console.WriteLine("Press the ENTER key to cancel...\n");

    Task cancelTask = Task.Run(() =>
    {
        while (Console.ReadKey().Key != ConsoleKey.Enter)
        {
            Console.WriteLine("Press the ENTER key to cancel...");
        }

        Console.WriteLine("\nENTER key pressed: cancelling downloads.\n");
        s_cts.Cancel();
    });

    Task sumPageSizesTask = SumPageSizesAsync();

    await Task.WhenAny(new[] { cancelTask, sumPageSizesTask });

    Console.WriteLine("Application ending.");
}

```

The updated `Main` method is now considered an [Async main](#), which allows for an asynchronous entry point into the executable. It writes a few instructional messages to the console, then declares a `Task` instance named `cancelTask`, which will read console key strokes. If the Enter key is pressed, a call to [CancellationTokenSource.Cancel\(\)](#) is made. This will signal cancellation. Next, the `sumPageSizesTask` variable is assigned from the `SumPageSizesAsync` method. Both tasks are then passed to [Task.WhenAny\(Task\[\]\)](#), which will continue when any of the two tasks have completed.

Create the asynchronous sum page sizes method

Below the `Main` method, add the `SumPageSizesAsync` method:

```
static async Task SumPageSizesAsync()
{
    var stopwatch = Stopwatch.StartNew();

    int total = 0;
    foreach (string url in s_urlList)
    {
        int contentLength = await ProcessUrlAsync(url, s_client, s_cts.Token);
        total += contentLength;
    }

    stopwatch.Stop();

    Console.WriteLine($"Total bytes returned: {total:#,#}");
    Console.WriteLine($"Elapsed time: {stopwatch.Elapsed}\n");
}
```

The method starts by instantiating and starting a [Stopwatch](#). It then loops through each URL in the `s_urlList` and calls `ProcessUrlAsync`. With each iteration, the `s_cts.Token` is passed into the `ProcessUrlAsync` method and the code returns a `Task<TResult>`, where `TResult` is an integer:

```
int total = 0;
foreach (string url in s_urlList)
{
    int contentLength = await ProcessUrlAsync(url, s_client, s_cts.Token);
    total += contentLength;
}
```

Add process method

Add the following `ProcessUrlAsync` method below the `SumPageSizesAsync` method:

```
static async Task<int> ProcessUrlAsync(string url, HttpClient client, CancellationToken token)
{
    HttpResponseMessage response = await client.GetAsync(url, token);
    byte[] content = await response.Content.ReadAsByteArrayAsync(token);
    Console.WriteLine($"{url,-60} {content.Length,10:#,#}");

    return content.Length;
}
```

For any given URL, the method will use the `client` instance provided to get the response as a `byte[]`. The [CancellationToken](#) instance is passed into the [HttpClient.GetAsync\(String, CancellationToken\)](#) and [HttpContent.ReadAsByteArrayAsync\(\)](#) methods. The `token` is used to register for requested cancellation. The length is returned after the URL and length is written to the console.

Example application output

```
Application started.
Press the ENTER key to cancel...

https://docs.microsoft.com                37,357
https://docs.microsoft.com/aspnet/core    85,589
https://docs.microsoft.com/azure          398,939
https://docs.microsoft.com/azure/devops   73,663
https://docs.microsoft.com/dotnet         67,452
https://docs.microsoft.com/dynamics365    48,582
https://docs.microsoft.com/education      22,924

ENTER key pressed: cancelling downloads.

Application ending.
```

Complete example

The following code is the complete text of the *Program.cs* file for the example.

```
using System;
using System.Collections.Generic;
using System.Diagnostics;
using System.Net.Http;
using System.Threading;
using System.Threading.Tasks;

class Program
{
    static readonly CancellationTokenSource s_cts = new CancellationTokenSource();

    static readonly HttpClient s_client = new HttpClient
    {
        MaxResponseContentBufferSize = 1_000_000
    };

    static readonly IEnumerable<string> s_urllist = new string[]
    {
        "https://docs.microsoft.com",
        "https://docs.microsoft.com/aspnet/core",
        "https://docs.microsoft.com/azure",
        "https://docs.microsoft.com/azure/devops",
        "https://docs.microsoft.com/dotnet",
        "https://docs.microsoft.com/dynamics365",
        "https://docs.microsoft.com/education",
        "https://docs.microsoft.com/enterprise-mobility-security",
        "https://docs.microsoft.com/gaming",
        "https://docs.microsoft.com/graph",
        "https://docs.microsoft.com/microsoft-365",
        "https://docs.microsoft.com/office",
        "https://docs.microsoft.com/powershell",
        "https://docs.microsoft.com/sql",
        "https://docs.microsoft.com/surface",
        "https://docs.microsoft.com/system-center",
        "https://docs.microsoft.com/visualstudio",
        "https://docs.microsoft.com/windows",
        "https://docs.microsoft.com/xamarin"
    };

    static async Task Main()
    {
        Console.WriteLine("Application started.");
        Console.WriteLine("Press the ENTER key to cancel...\n");
```

```

Task cancelTask = Task.Run(() =>
{
    while (Console.ReadKey().Key != ConsoleKey.Enter)
    {
        Console.WriteLine("Press the ENTER key to cancel...");
    }

    Console.WriteLine("\nENTER key pressed: cancelling downloads.\n");
    s_cts.Cancel();
});

Task sumPageSizesTask = SumPageSizesAsync();

await Task.WhenAny(new[] { cancelTask, sumPageSizesTask });

Console.WriteLine("Application ending.");
}

static async Task SumPageSizesAsync()
{
    var stopwatch = Stopwatch.StartNew();

    int total = 0;
    foreach (string url in s_urlList)
    {
        int contentLength = await ProcessUrlAsync(url, s_client, s_cts.Token);
        total += contentLength;
    }

    stopwatch.Stop();

    Console.WriteLine($"Total bytes returned: {total:#,##}");
    Console.WriteLine($"Elapsed time: {stopwatch.Elapsed}\n");
}

static async Task<int> ProcessUrlAsync(string url, HttpClient client, CancellationToken token)
{
    HttpResponseMessage response = await client.GetAsync(url, token);
    byte[] content = await response.Content.ReadAsByteArrayAsync(token);
    Console.WriteLine($"{url,-60} {content.Length,10:#,##}");

    return content.Length;
}
}

```

See also

- [CancellationToken](#)
- [CancellationTokenSource](#)
- [Asynchronous programming with async and await \(C#\)](#)

Next steps

[Cancel async tasks after a period of time \(C#\)](#)

Cancel async tasks after a period of time (C#)

12/28/2021 • 2 minutes to read • [Edit Online](#)

You can cancel an asynchronous operation after a period of time by using the [CancellationTokenSource.CancelAfter](#) method if you don't want to wait for the operation to finish. This method schedules the cancellation of any associated tasks that aren't complete within the period of time that's designated by the `CancelAfter` expression.

This example adds to the code that's developed in [Cancel a list of tasks \(C#\)](#) to download a list of websites and to display the length of the contents of each one.

This tutorial covers:

- Updating an existing .NET console application
- Scheduling a cancellation

Prerequisites

This tutorial requires the following:

- You're expected to have created an application in the [Cancel a list of tasks \(C#\)](#) tutorial
- [.NET 5 or later SDK](#)
- Integrated development environment (IDE)
 - [We recommend Visual Studio, Visual Studio Code, or Visual Studio for Mac](#)

Update application entry point

Replace the existing `Main` method with the following:

```
static async Task Main()
{
    Console.WriteLine("Application started.");

    try
    {
        s_cts.CancelAfter(3500);

        await SumPageSizesAsync();
    }
    catch (OperationCanceledException)
    {
        Console.WriteLine("\nTasks cancelled: timed out.\n");
    }
    finally
    {
        s_cts.Dispose();
    }

    Console.WriteLine("Application ending.");
}
```

The updated `Main` method writes a few instructional messages to the console. Within the `try catch`, a call to [CancellationTokenSource.CancelAfter\(Int32\)](#) schedules a cancellation. This will signal cancellation after a period of time.

Next, the `SumPageSizesAsync` method is awaited. If processing all of the URLs occurs faster than the scheduled cancellation, the application ends. However, if the scheduled cancellation is triggered before all of the URLs are processed, a `OperationCanceledException` is thrown.

Example application output

```
Application started.

https://docs.microsoft.com           37,357
https://docs.microsoft.com/aspnet/core 85,589
https://docs.microsoft.com/azure      398,939
https://docs.microsoft.com/azure/devops 73,663

Tasks cancelled: timed out.

Application ending.
```

Complete example

The following code is the complete text of the *Program.cs* file for the example.

```
using System;
using System.Collections.Generic;
using System.Diagnostics;
using System.Net.Http;
using System.Threading;
using System.Threading.Tasks;

class Program
{
    static readonly CancellationTokenSource s_cts = new CancellationTokenSource();

    static readonly HttpClient s_client = new HttpClient
    {
        MaxResponseContentBufferSize = 1_000_000
    };

    static readonly IEnumerable<string> s_urlList = new string[]
    {
        "https://docs.microsoft.com",
        "https://docs.microsoft.com/aspnet/core",
        "https://docs.microsoft.com/azure",
        "https://docs.microsoft.com/azure/devops",
        "https://docs.microsoft.com/dotnet",
        "https://docs.microsoft.com/dynamics365",
        "https://docs.microsoft.com/education",
        "https://docs.microsoft.com/enterprise-mobility-security",
        "https://docs.microsoft.com/gaming",
        "https://docs.microsoft.com/graph",
        "https://docs.microsoft.com/microsoft-365",
        "https://docs.microsoft.com/office",
        "https://docs.microsoft.com/powershell",
        "https://docs.microsoft.com/sql",
        "https://docs.microsoft.com/surface",
        "https://docs.microsoft.com/system-center",
        "https://docs.microsoft.com/visualstudio",
        "https://docs.microsoft.com/windows",
        "https://docs.microsoft.com/xamarin"
    };

    static async Task Main()
    {
        Console.WriteLine("Application started.");
```

```

        try
        {
            s_cts.CancelAfter(3500);

            await SumPageSizesAsync();
        }
        catch (OperationCanceledException)
        {
            Console.WriteLine("\nTasks cancelled: timed out.\n");
        }
        finally
        {
            s_cts.Dispose();
        }

        Console.WriteLine("Application ending.");
    }

    static async Task SumPageSizesAsync()
    {
        var stopwatch = Stopwatch.StartNew();

        int total = 0;
        foreach (string url in s_urlList)
        {
            int contentLength = await ProcessUrlAsync(url, s_client, s_cts.Token);
            total += contentLength;
        }

        stopwatch.Stop();

        Console.WriteLine($"Total bytes returned: {total:#,##}");
        Console.WriteLine($"Elapsed time: {stopwatch.Elapsed}\n");
    }

    static async Task<int> ProcessUrlAsync(string url, HttpClient client, CancellationToken token)
    {
        HttpResponseMessage response = await client.GetAsync(url, token);
        byte[] content = await response.Content.ReadAsByteArrayAsync(token);
        Console.WriteLine($"{url,-60} {content.Length,10:#,##}");

        return content.Length;
    }
}

```

See also

- [CancellationToken](#)
- [CancellationTokenSource](#)
- [Asynchronous programming with async and await \(C#\)](#)
- [Cancel a list of tasks \(C#\)](#)

Process asynchronous tasks as they complete (C#)

12/28/2021 • 9 minutes to read • [Edit Online](#)

By using [Task.WhenAny](#), you can start multiple tasks at the same time and process them one by one as they're completed rather than process them in the order in which they're started.

The following example uses a query to create a collection of tasks. Each task downloads the contents of a specified website. In each iteration of a while loop, an awaited call to [WhenAny](#) returns the task in the collection of tasks that finishes its download first. That task is removed from the collection and processed. The loop repeats until the collection contains no more tasks.

Prerequisites

You can follow this tutorial by using one of the following options:

- [Visual Studio 2022 version 17.0.0 Preview](#) with the **.NET desktop development** workload installed. The .NET 6.0 SDK is automatically installed when you select this workload.
- The [.NET 6.0 SDK](#) with a code editor of your choice, such as [Visual Studio Code](#).

Create example application

Create a new .NET Core console application that targets .NET 6.0. You can create one by using the [dotnet new console](#) command or from Visual Studio.

Open the *Program.cs* file in your code editor, and replace the existing code with this code:

```
using System.Diagnostics;

namespace ProcessTasksAsTheyFinish;

class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine("Hello World!");
    }
}
```

Add fields

In the `Program` class definition, add the following two fields:

```

static readonly HttpClient s_client = new HttpClient
{
    MaxResponseContentBufferSize = 1_000_000
};

static readonly IEnumerable<string> s_urllist = new string[]
{
    "https://docs.microsoft.com",
    "https://docs.microsoft.com/aspnet/core",
    "https://docs.microsoft.com/azure",
    "https://docs.microsoft.com/azure/devops",
    "https://docs.microsoft.com/dotnet",
    "https://docs.microsoft.com/dynamics365",
    "https://docs.microsoft.com/education",
    "https://docs.microsoft.com/enterprise-mobility-security",
    "https://docs.microsoft.com/gaming",
    "https://docs.microsoft.com/graph",
    "https://docs.microsoft.com/microsoft-365",
    "https://docs.microsoft.com/office",
    "https://docs.microsoft.com/powershell",
    "https://docs.microsoft.com/sql",
    "https://docs.microsoft.com/surface",
    "https://docs.microsoft.com/system-center",
    "https://docs.microsoft.com/visualstudio",
    "https://docs.microsoft.com/windows",
    "https://docs.microsoft.com/xamarin"
};

```

The `HttpClient` exposes the ability to send HTTP requests and receive HTTP responses. The `s_urllist` holds all of the URLs that the application plans to process.

Update application entry point

The main entry point into the console application is the `Main` method. Replace the existing method with the following:

```

static Task Main() => SumPageSizesAsync();

```

The updated `Main` method is now considered an [Async main](#), which allows for an asynchronous entry point into the executable. It is expressed as a call to `SumPageSizesAsync`.

Create the asynchronous sum page sizes method

Below the `Main` method, add the `SumPageSizesAsync` method:

```

static async Task SumPageSizesAsync()
{
    var stopwatch = Stopwatch.StartNew();

    IEnumerable<Task<int>> downloadTasksQuery =
        from url in s_urllist
        select ProcessUrlAsync(url, s_client);

    List<Task<int>> downloadTasks = downloadTasksQuery.ToList();

    int total = 0;
    while (downloadTasks.Any())
    {
        Task<int> finishedTask = await Task.WhenAny(downloadTasks);
        downloadTasks.Remove(finishedTask);
        total += await finishedTask;
    }

    stopwatch.Stop();

    Console.WriteLine($"Total bytes returned: {total:#,#}");
    Console.WriteLine($"Elapsed time: {stopwatch.Elapsed}\n");
}

```

The method starts by instantiating and starting a [Stopwatch](#). It then includes a query that, when executed, creates a collection of tasks. Each call to `ProcessUrlAsync` in the following code returns a `Task<TResult>`, where `TResult` is an integer:

```

IEnumerable<Task<int>> downloadTasksQuery =
    from url in s_urllist
    select ProcessUrlAsync(url, s_client);

```

Due to [deferred execution](#) with the LINQ, you call `Enumerable.ToList` to start each task.

```

List<Task<int>> downloadTasks = downloadTasksQuery.ToList();

```

The `while` loop performs the following steps for each task in the collection:

1. Awaits a call to `WhenAny` to identify the first task in the collection that has finished its download.

```

Task<int> finishedTask = await Task.WhenAny(downloadTasks);

```

2. Removes that task from the collection.

```

downloadTasks.Remove(finishedTask);

```

3. Awaits `finishedTask`, which is returned by a call to `ProcessUrlAsync`. The `finishedTask` variable is a `Task<TResult>` where `TResult` is an integer. The task is already complete, but you await it to retrieve the length of the downloaded website, as the following example shows. If the task is faulted, `await` will throw the first child exception stored in the `AggregateException`, unlike reading the `Task<TResult>.Result` property, which would throw the `AggregateException`.

```

total += await finishedTask;

```

Add process method

Add the following `ProcessUrlAsync` method below the `SumPageSizesAsync` method:

```
static async Task<int> ProcessUrlAsync(string url, HttpClient client)
{
    byte[] content = await client.GetByteArrayAsync(url);
    Console.WriteLine($"{url,-60} {content.Length,10:#,#}");

    return content.Length;
}
```

For any given URL, the method will use the `client` instance provided to get the response as a `byte[]`. The length is returned after the URL and length is written to the console.

Run the program several times to verify that the downloaded lengths don't always appear in the same order.

Caution

You can use `WhenAny` in a loop, as described in the example, to solve problems that involve a small number of tasks. However, other approaches are more efficient if you have a large number of tasks to process. For more information and examples, see [Processing tasks as they complete](#).

Complete example

The following code is the complete text of the *Program.cs* file for the example.

```
using System.Diagnostics;

namespace ProcessTasksAsTheyFinish;

class Program
{
    static readonly HttpClient s_client = new HttpClient
    {
        MaxResponseContentBufferSize = 1_000_000
    };

    static readonly IEnumerable<string> s_urlList = new string[]
    {
        "https://docs.microsoft.com",
        "https://docs.microsoft.com/aspnet/core",
        "https://docs.microsoft.com/azure",
        "https://docs.microsoft.com/azure/devops",
        "https://docs.microsoft.com/dotnet",
        "https://docs.microsoft.com/dynamics365",
        "https://docs.microsoft.com/education",
        "https://docs.microsoft.com/enterprise-mobility-security",
        "https://docs.microsoft.com/gaming",
        "https://docs.microsoft.com/graph",
        "https://docs.microsoft.com/microsoft-365",
        "https://docs.microsoft.com/office",
        "https://docs.microsoft.com/powershell",
        "https://docs.microsoft.com/sql",
        "https://docs.microsoft.com/surface",
        "https://docs.microsoft.com/system-center",
        "https://docs.microsoft.com/visualstudio",
        "https://docs.microsoft.com/windows",
        "https://docs.microsoft.com/xamarin"
    };

    static Task Main() => SumPageSizesAsync();

    static async Task SumPageSizesAsync()
    {
        r
```

```

1
    var stopwatch = Stopwatch.StartNew();

    IEnumerable<Task<int>> downloadTasksQuery =
        from url in s_urlList
        select ProcessUrlAsync(url, s_client);

    List<Task<int>> downloadTasks = downloadTasksQuery.ToList();

    int total = 0;
    while (downloadTasks.Any())
    {
        Task<int> finishedTask = await Task.WhenAny(downloadTasks);
        downloadTasks.Remove(finishedTask);
        total += await finishedTask;
    }

    stopwatch.Stop();

    Console.WriteLine($"Total bytes returned: {total:#,##}");
    Console.WriteLine($"Elapsed time: {stopwatch.Elapsed}\n");
}

static async Task<int> ProcessUrlAsync(string url, HttpClient client)
{
    byte[] content = await client.GetByteArrayAsync(url);
    Console.WriteLine($"{url,-60} {content.Length,10:#,##}");

    return content.Length;
}
}

// Example output:
// https://docs.microsoft.com/windows                25,513
// https://docs.microsoft.com/gaming                  30,705
// https://docs.microsoft.com/dotnet                  69,626
// https://docs.microsoft.com/dynamics365             50,756
// https://docs.microsoft.com/surface                  35,519
// https://docs.microsoft.com                          39,531
// https://docs.microsoft.com/azure/devops             75,837
// https://docs.microsoft.com/xamarin                  60,284
// https://docs.microsoft.com/system-center            43,444
// https://docs.microsoft.com/enterprise-mobility-security 28,946
// https://docs.microsoft.com/microsoft-365           43,278
// https://docs.microsoft.com/visualstudio            31,414
// https://docs.microsoft.com/office                  42,292
// https://docs.microsoft.com/azure                   401,113
// https://docs.microsoft.com/graph                   46,831
// https://docs.microsoft.com/education               25,098
// https://docs.microsoft.com/powershell              58,173
// https://docs.microsoft.com/aspnet/core             87,763
// https://docs.microsoft.com/sql                     53,362

// Total bytes returned: 1,249,485
// Elapsed time: 00:00:02.7068725

```

See also

- [WhenAny](#)
- [Asynchronous programming with async and await \(C#\)](#)

By using [Task.WhenAny](#), you can start multiple tasks at the same time and process them one by one as they're completed rather than process them in the order in which they're started.

The following example uses a query to create a collection of tasks. Each task downloads the contents of a specified website. In each iteration of a while loop, an awaited call to [WhenAny](#) returns the task in the collection

of tasks that finishes its download first. That task is removed from the collection and processed. The loop repeats until the collection contains no more tasks.

Prerequisites

You can follow this tutorial by using one of the following options:

- [Visual Studio](#) with the **.NET desktop development** workload installed. The .NET SDK is automatically installed when you select this workload.
- The [.NET 5.0 SDK](#) with a code editor of your choice, such as [Visual Studio Code](#).

Create example application

Create a new .NET Core console application that targets .NET 5.0 or .NET Core 3.1. You can create one by using the `dotnet new console` command or from Visual Studio. Open the *Program.cs* file in your favorite code editor.

Replace using statements

Replace the existing using statements with these declarations:

```
using System;
using System.Collections.Generic;
using System.Diagnostics;
using System.Linq;
using System.Net.Http;
using System.Threading.Tasks;
```

Add fields

In the `Program` class definition, add the following two fields:

```
static readonly HttpClient s_client = new HttpClient
{
    MaxResponseContentBufferSize = 1_000_000
};

static readonly IEnumerable<string> s_urlList = new string[]
{
    "https://docs.microsoft.com",
    "https://docs.microsoft.com/aspnet/core",
    "https://docs.microsoft.com/azure",
    "https://docs.microsoft.com/azure/devops",
    "https://docs.microsoft.com/dotnet",
    "https://docs.microsoft.com/dynamics365",
    "https://docs.microsoft.com/education",
    "https://docs.microsoft.com/enterprise-mobility-security",
    "https://docs.microsoft.com/gaming",
    "https://docs.microsoft.com/graph",
    "https://docs.microsoft.com/microsoft-365",
    "https://docs.microsoft.com/office",
    "https://docs.microsoft.com/powershell",
    "https://docs.microsoft.com/sql",
    "https://docs.microsoft.com/surface",
    "https://docs.microsoft.com/system-center",
    "https://docs.microsoft.com/visualstudio",
    "https://docs.microsoft.com/windows",
    "https://docs.microsoft.com/xamarin"
};
```

The `HttpClient` exposes the ability to send HTTP requests and receive HTTP responses. The `s_urlList` holds all

of the URLs that the application plans to process.

Update application entry point

The main entry point into the console application is the `Main` method. Replace the existing method with the following:

```
static Task Main() => SumPageSizesAsync();
```

The updated `Main` method is now considered an [Async main](#), which allows for an asynchronous entry point into the executable. It is expressed a call to `SumPageSizesAsync`.

Create the asynchronous sum page sizes method

Below the `Main` method, add the `SumPageSizesAsync` method:

```
static async Task SumPageSizesAsync()
{
    var stopwatch = Stopwatch.StartNew();

    IEnumerable<Task<int>> downloadTasksQuery =
        from url in s_urllist
        select ProcessUrlAsync(url, s_client);

    List<Task<int>> downloadTasks = downloadTasksQuery.ToList();

    int total = 0;
    while (downloadTasks.Any())
    {
        Task<int> finishedTask = await Task.WhenAny(downloadTasks);
        downloadTasks.Remove(finishedTask);
        total += await finishedTask;
    }

    stopwatch.Stop();

    Console.WriteLine($"Total bytes returned: {total:#,#}");
    Console.WriteLine($"Elapsed time: {stopwatch.Elapsed}\n");
}
```

The method starts by instantiating and starting a [Stopwatch](#). It then includes a query that, when executed, creates a collection of tasks. Each call to `ProcessUrlAsync` in the following code returns a `Task<TResult>`, where `TResult` is an integer:

```
IEnumerable<Task<int>> downloadTasksQuery =
    from url in s_urllist
    select ProcessUrlAsync(url, s_client);
```

Due to [deferred execution](#) with the LINQ, you call `Enumerable.ToList` to start each task.

```
List<Task<int>> downloadTasks = downloadTasksQuery.ToList();
```

The `while` loop performs the following steps for each task in the collection:

1. Awaits a call to `WhenAny` to identify the first task in the collection that has finished its download.

```
Task<int> finishedTask = await Task.WhenAny(downloadTasks);
```

2. Removes that task from the collection.

```
downloadTasks.Remove(finishedTask);
```

3. Awaits `finishedTask`, which is returned by a call to `ProcessUrlAsync`. The `finishedTask` variable is a `Task<TResult>` where `TResult` is an integer. The task is already complete, but you await it to retrieve the length of the downloaded website, as the following example shows. If the task is faulted, `await` will throw the first child exception stored in the `AggregateException`, unlike reading the `Task<TResult>.Result` property, which would throw the `AggregateException`.

```
total += await finishedTask;
```

Add process method

Add the following `ProcessUrlAsync` method below the `SumPageSizesAsync` method:

```
static async Task<int> ProcessUrlAsync(string url, HttpClient client)
{
    byte[] content = await client.GetByteArrayAsync(url);
    Console.WriteLine($"{url,-60} {content.Length,10:#,#}");

    return content.Length;
}
```

For any given URL, the method will use the `client` instance provided to get the response as a `byte[]`. The length is returned after the URL and length is written to the console.

Run the program several times to verify that the downloaded lengths don't always appear in the same order.

Caution

You can use `WhenAny` in a loop, as described in the example, to solve problems that involve a small number of tasks. However, other approaches are more efficient if you have a large number of tasks to process. For more information and examples, see [Processing tasks as they complete](#).

Complete example

The following code is the complete text of the *Program.cs* file for the example.

```
using System;
using System.Collections.Generic;
using System.Diagnostics;
using System.Linq;
using System.Net.Http;
using System.Threading.Tasks;

namespace ProcessTasksAsTheyFinish
{
    class Program
    {
        static readonly HttpClient s_client = new HttpClient
        {
            MaxResponseContentBufferSize = 1_000_000
        };
    }
}
```

```

static readonly IEnumerable<string> s_urlList = new string[]
{
    "https://docs.microsoft.com",
    "https://docs.microsoft.com/aspnet/core",
    "https://docs.microsoft.com/azure",
    "https://docs.microsoft.com/azure/devops",
    "https://docs.microsoft.com/dotnet",
    "https://docs.microsoft.com/dynamics365",
    "https://docs.microsoft.com/education",
    "https://docs.microsoft.com/enterprise-mobility-security",
    "https://docs.microsoft.com/gaming",
    "https://docs.microsoft.com/graph",
    "https://docs.microsoft.com/microsoft-365",
    "https://docs.microsoft.com/office",
    "https://docs.microsoft.com/powershell",
    "https://docs.microsoft.com/sql",
    "https://docs.microsoft.com/surface",
    "https://docs.microsoft.com/system-center",
    "https://docs.microsoft.com/visualstudio",
    "https://docs.microsoft.com/windows",
    "https://docs.microsoft.com/xamarin"
};

static Task Main() => SumPageSizesAsync();

static async Task SumPageSizesAsync()
{
    var stopwatch = Stopwatch.StartNew();

    IEnumerable<Task<int>> downloadTasksQuery =
        from url in s_urlList
        select ProcessUrlAsync(url, s_client);

    List<Task<int>> downloadTasks = downloadTasksQuery.ToList();

    int total = 0;
    while (downloadTasks.Any())
    {
        Task<int> finishedTask = await Task.WhenAny(downloadTasks);
        downloadTasks.Remove(finishedTask);
        total += await finishedTask;
    }

    stopwatch.Stop();

    Console.WriteLine($"Total bytes returned: {total:#,##}");
    Console.WriteLine($"Elapsed time: {stopwatch.Elapsed}\n");
}

static async Task<int> ProcessUrlAsync(string url, HttpClient client)
{
    byte[] content = await client.GetByteArrayAsync(url);
    Console.WriteLine($"{url,-60} {content.Length,10:#,##}");

    return content.Length;
}
}

// Example output:
// https://docs.microsoft.com/windows                25,513
// https://docs.microsoft.com/gaming                 30,705
// https://docs.microsoft.com/dotnet                 69,626
// https://docs.microsoft.com/dynamics365            50,756
// https://docs.microsoft.com/surface                35,519
// https://docs.microsoft.com                       39,531
// https://docs.microsoft.com/azure/devops           75,837
// https://docs.microsoft.com/xamarin                60,284
// https://docs.microsoft.com/system-center          43,444
// https://docs.microsoft.com/enterprise-mobility-security 28,946

```

```
// https://docs.microsoft.com/microsoft-365 43,278
// https://docs.microsoft.com/visualstudio 31,414
// https://docs.microsoft.com/office 42,292
// https://docs.microsoft.com/azure 401,113
// https://docs.microsoft.com/graph 46,831
// https://docs.microsoft.com/education 25,098
// https://docs.microsoft.com/powershell 58,173
// https://docs.microsoft.com/aspnet/core 87,763
// https://docs.microsoft.com/sql 53,362

// Total bytes returned: 1,249,485
// Elapsed time: 00:00:02.7068725
```

See also

- [WhenAny](#)
- [Asynchronous programming with async and await \(C#\)](#)

Asynchronous file access (C#)

12/28/2021 • 5 minutes to read • [Edit Online](#)

You can use the `async` feature to access files. By using the `async` feature, you can call into asynchronous methods without using callbacks or splitting your code across multiple methods or lambda expressions. To make synchronous code asynchronous, you just call an asynchronous method instead of a synchronous method and add a few keywords to the code.

You might consider the following reasons for adding asynchrony to file access calls:

- Asynchrony makes UI applications more responsive because the UI thread that launches the operation can perform other work. If the UI thread must execute code that takes a long time (for example, more than 50 milliseconds), the UI may freeze until the I/O is complete and the UI thread can again process keyboard and mouse input and other events.
- Asynchrony improves the scalability of ASP.NET and other server-based applications by reducing the need for threads. If the application uses a dedicated thread per response and a thousand requests are being handled simultaneously, a thousand threads are needed. Asynchronous operations often don't need to use a thread during the wait. They use the existing I/O completion thread briefly at the end.
- The latency of a file access operation might be very low under current conditions, but the latency may greatly increase in the future. For example, a file may be moved to a server that's across the world.
- The added overhead of using the `Async` feature is small.
- Asynchronous tasks can easily be run in parallel.

Use appropriate classes

The simple examples in this topic demonstrate [File.WriteAllTextAsync](#) and [File.ReadAllTextAsync](#). For finite control over the file I/O operations, use the [FileStream](#) class, which has an option that causes asynchronous I/O to occur at the operating system level. By using this option, you can avoid blocking a thread pool thread in many cases. To enable this option, you specify the `useAsync=true` or `options=FileOptions.Asynchronous` argument in the constructor call.

You can't use this option with [StreamReader](#) and [StreamWriter](#) if you open them directly by specifying a file path. However, you can use this option if you provide them a [Stream](#) that the [FileStream](#) class opened.

Asynchronous calls are faster in UI apps even if a thread pool thread is blocked, because the UI thread isn't blocked during the wait.

Write text

The following examples write text to a file. At each `await` statement, the method immediately exits. When the file I/O is complete, the method resumes at the statement that follows the `await` statement. The `async` modifier is in the definition of methods that use the `await` statement.

Simple example

```
public async Task SimpleWriteAsync()
{
    string filePath = "simple.txt";
    string text = $"Hello World";

    await File.WriteAllTextAsync(filePath, text);
}
```

Finite control example

```
public async Task ProcessWriteAsync()
{
    string filePath = "temp.txt";
    string text = $"Hello World{Environment.NewLine}";

    await WriteTextAsync(filePath, text);
}

async Task WriteTextAsync(string filePath, string text)
{
    byte[] encodedText = Encoding.Unicode.GetBytes(text);

    using var sourceStream =
        new FileStream(
            filePath,
            FileMode.Create, FileAccess.Write, FileShare.None,
            bufferSize: 4096, useAsync: true);

    await sourceStream.WriteAsync(encodedText, 0, encodedText.Length);
}
```

The original example has the statement `await sourceStream.WriteAsync(encodedText, 0, encodedText.Length);`, which is a contraction of the following two statements:

```
Task theTask = sourceStream.WriteAsync(encodedText, 0, encodedText.Length);
await theTask;
```

The first statement returns a task and causes file processing to start. The second statement with the `await` causes the method to immediately exit and return a different task. When the file processing later completes, execution returns to the statement that follows the `await`.

Read text

The following examples read text from a file.

Simple example

```
public async Task SimpleReadAsync()
{
    string filePath = "simple.txt";
    string text = await File.ReadAllTextAsync(filePath);

    Console.WriteLine(text);
}
```

Finite control example

The text is buffered and, in this case, placed into a [StringBuilder](#). Unlike in the previous example, the evaluation of the `await` produces a value. The [ReadAsync](#) method returns a `Task<Int32>`, so the evaluation of the `await` produces an `Int32` value `numRead` after the operation completes. For more information, see [Async Return Types \(C#\)](#).

```

public async Task ProcessReadAsync()
{
    try
    {
        string filePath = "temp.txt";
        if (File.Exists(filePath) != false)
        {
            string text = await ReadTextAsync(filePath);
            Console.WriteLine(text);
        }
        else
        {
            Console.WriteLine($"file not found: {filePath}");
        }
    }
    catch (Exception ex)
    {
        Console.WriteLine(ex.Message);
    }
}

async Task<string> ReadTextAsync(string filePath)
{
    using var sourceStream =
        new FileStream(
            filePath,
            FileMode.Open, FileAccess.Read, FileShare.Read,
            bufferSize: 4096, useAsync: true);

    var sb = new StringBuilder();

    byte[] buffer = new byte[0x1000];
    int numRead;
    while ((numRead = await sourceStream.ReadAsync(buffer, 0, buffer.Length)) != 0)
    {
        string text = Encoding.Unicode.GetString(buffer, 0, numRead);
        sb.Append(text);
    }

    return sb.ToString();
}

```

Parallel asynchronous I/O

The following examples demonstrate parallel processing by writing 10 text files.

Simple example

```

public async Task SimpleParallelWriteAsync()
{
    string folder = Directory.CreateDirectory("tempfolder").Name;
    IList<Task> writeTaskList = new List<Task>();

    for (int index = 11; index <= 20; ++ index)
    {
        string fileName = $"file-{index:00}.txt";
        string filePath = $"{folder}/{fileName}";
        string text = $"In file {index}{Environment.NewLine}";

        writeTaskList.Add(File.WriteAllTextAsync(filePath, text));
    }

    await Task.WhenAll(writeTaskList);
}

```


Finite control example

For each file, the [WriteAsync](#) method returns a task that is then added to a list of tasks. The

`await Task.WhenAll(tasks);` statement exits the method and resumes within the method when file processing is complete for all of the tasks.

The example closes all [FileStream](#) instances in a `finally` block after the tasks are complete. If each `FileStream` was instead created in a `using` statement, the `FileStream` might be disposed of before the task was complete.

Any performance boost is almost entirely from the parallel processing and not the asynchronous processing. The advantages of asynchrony are that it doesn't tie up multiple threads, and that it doesn't tie up the user interface thread.

```
public async Task ProcessMultipleWritesAsync()
{
    IList<FileStream> sourceStreams = new List<FileStream>();

    try
    {
        string folder = Directory.CreateDirectory("tempfolder").Name;
        IList<Task> writeTaskList = new List<Task>();

        for (int index = 1; index <= 10; ++ index)
        {
            string fileName = $"file-{index:00}.txt";
            string filePath = $"{folder}/{fileName}";

            string text = $"In file {index}{Environment.NewLine}";
            byte[] encodedText = Encoding.Unicode.GetBytes(text);

            var sourceStream =
                new FileStream(
                    filePath,
                    FileMode.Create, FileAccess.Write, FileShare.None,
                    bufferSize: 4096, useAsync: true);

            Task writeTask = sourceStream.WriteAsync(encodedText, 0, encodedText.Length);
            sourceStreams.Add(sourceStream);

            writeTaskList.Add(writeTask);
        }

        await Task.WhenAll(writeTaskList);
    }
    finally
    {
        foreach (FileStream sourceStream in sourceStreams)
        {
            sourceStream.Close();
        }
    }
}
```

When using the [WriteAsync](#) and [ReadAsync](#) methods, you can specify a [CancellationToken](#), which you can use to cancel the operation mid-stream. For more information, see [Cancellation in managed threads](#).

See also

- [Asynchronous programming with async and await \(C#\)](#)
- [Async return types \(C#\)](#)

Attributes (C#)

12/28/2021 • 5 minutes to read • [Edit Online](#)

Attributes provide a powerful method of associating metadata, or declarative information, with code (assemblies, types, methods, properties, and so forth). After an attribute is associated with a program entity, the attribute can be queried at run time by using a technique called *reflection*. For more information, see [Reflection \(C#\)](#).

Attributes have the following properties:

- Attributes add metadata to your program. *Metadata* is information about the types defined in a program. All .NET assemblies contain a specified set of metadata that describes the types and type members defined in the assembly. You can add custom attributes to specify any additional information that is required. For more information, see, [Creating Custom Attributes \(C#\)](#).
- You can apply one or more attributes to entire assemblies, modules, or smaller program elements such as classes and properties.
- Attributes can accept arguments in the same way as methods and properties.
- Your program can examine its own metadata or the metadata in other programs by using reflection. For more information, see [Accessing Attributes by Using Reflection \(C#\)](#).

Using attributes

Attributes can be placed on almost any declaration, though a specific attribute might restrict the types of declarations on which it is valid. In C#, you specify an attribute by placing the name of the attribute enclosed in square brackets ([]) above the declaration of the entity to which it applies.

In this example, the [SerializableAttribute](#) attribute is used to apply a specific characteristic to a class:

```
[Serializable]
public class SampleClass
{
    // Objects of this type can be serialized.
}
```

A method with the attribute [DllImportAttribute](#) is declared like the following example:

```
[System.Runtime.InteropServices.DllImport("user32.dll")]
extern static void SampleMethod();
```

More than one attribute can be placed on a declaration as the following example shows:

```
using System.Runtime.InteropServices;
```

```
void MethodA([In][Out] ref double x) { }
void MethodB([Out][In] ref double x) { }
void MethodC([In, Out] ref double x) { }
```

Some attributes can be specified more than once for a given entity. An example of such a multiuse attribute is [ConditionalAttribute](#):

```
[Conditional("DEBUG"), Conditional("TEST1")]
void TraceMethod()
{
    // ...
}
```

NOTE

By convention, all attribute names end with the word "Attribute" to distinguish them from other items in the .NET libraries. However, you do not need to specify the attribute suffix when using attributes in code. For example, `[DllImport]` is equivalent to `[DllImportAttribute]`, but `DllImportAttribute` is the attribute's actual name in the .NET Class Library.

Attribute parameters

Many attributes have parameters, which can be positional, unnamed, or named. Any positional parameters must be specified in a certain order and cannot be omitted. Named parameters are optional and can be specified in any order. Positional parameters are specified first. For example, these three attributes are equivalent:

```
[DllImport("user32.dll")]
[DllImport("user32.dll", SetLastError=false, ExactSpelling=false)]
[DllImport("user32.dll", ExactSpelling=false, SetLastError=false)]
```

The first parameter, the DLL name, is positional and always comes first; the others are named. In this case, both named parameters default to false, so they can be omitted. Positional parameters correspond to the parameters of the attribute constructor. Named or optional parameters correspond to either properties or fields of the attribute. Refer to the individual attribute's documentation for information on default parameter values.

For more information on allowed parameter types, see the [Attributes](#) section of the [C# language specification](#)

Attribute targets

The *target* of an attribute is the entity which the attribute applies to. For example, an attribute may apply to a class, a particular method, or an entire assembly. By default, an attribute applies to the element that follows it. But you can also explicitly identify, for example, whether an attribute is applied to a method, or to its parameter, or to its return value.

To explicitly identify an attribute target, use the following syntax:

```
[target : attribute-list]
```

The list of possible `target` values is shown in the following table.

TARGET VALUE	APPLIES TO
<code>assembly</code>	Entire assembly
<code>module</code>	Current assembly module
<code>field</code>	Field in a class or a struct
<code>event</code>	Event
<code>method</code>	Method or <code>get</code> and <code>set</code> property accessors

TARGET VALUE	APPLIES TO
<code>param</code>	Method parameters or <code>set</code> property accessor parameters
<code>property</code>	Property
<code>return</code>	Return value of a method, property indexer, or <code>get</code> property accessor
<code>type</code>	Struct, class, interface, enum, or delegate

You would specify the `field` target value to apply an attribute to the backing field created for an [auto-implemented property](#).

The following example shows how to apply attributes to assemblies and modules. For more information, see [Common Attributes \(C#\)](#).

```
using System;
using System.Reflection;
[assembly: AssemblyTitleAttribute("Production assembly 4")]
[module: CLSCompliant(true)]
```

The following example shows how to apply attributes to methods, method parameters, and method return values in C#.

```
// default: applies to method
[ValidatedContract]
int Method1() { return 0; }

// applies to method
[method: ValidatedContract]
int Method2() { return 0; }

// applies to parameter
int Method3([ValidatedContract] string contract) { return 0; }

// applies to return value
[return: ValidatedContract]
int Method4() { return 0; }
```

NOTE

Regardless of the targets on which `ValidatedContract` is defined to be valid, the `return` target has to be specified, even if `ValidatedContract` were defined to apply only to return values. In other words, the compiler will not use `AttributeUsage` information to resolve ambiguous attribute targets. For more information, see [AttributeUsage \(C#\)](#).

Common uses for attributes

The following list includes a few of the common uses of attributes in code:

- Marking methods using the `WebMethod` attribute in Web services to indicate that the method should be callable over the SOAP protocol. For more information, see [WebMethodAttribute](#).
- Describing how to marshal method parameters when interoperating with native code. For more information, see [MarshalAsAttribute](#).
- Describing the COM properties for classes, methods, and interfaces.

- Calling unmanaged code using the [DllImportAttribute](#) class.
- Describing your assembly in terms of title, version, description, or trademark.
- Describing which members of a class to serialize for persistence.
- Describing how to map between class members and XML nodes for XML serialization.
- Describing the security requirements for methods.
- Specifying characteristics used to enforce security.
- Controlling optimizations by the just-in-time (JIT) compiler so the code remains easy to debug.
- Obtaining information about the caller to a method.

Related sections

For more information, see:

- [Creating Custom Attributes \(C#\)](#)
- [Accessing Attributes by Using Reflection \(C#\)](#)
- [How to create a C/C++ union by using attributes \(C#\)](#)
- [Common Attributes \(C#\)](#)
- [Caller Information \(C#\)](#)

See also

- [C# Programming Guide](#)
- [Reflection \(C#\)](#)
- [Attributes](#)
- [Using Attributes in C#](#)

Creating Custom Attributes (C#)

12/28/2021 • 2 minutes to read • [Edit Online](#)

You can create your own custom attributes by defining an attribute class, a class that derives directly or indirectly from [Attribute](#), which makes identifying attribute definitions in metadata fast and easy. Suppose you want to tag types with the name of the programmer who wrote the type. You might define a custom [Author](#) attribute class:

```
[System.AttributeUsage(System.AttributeTargets.Class |
                      System.AttributeTargets.Struct)
]
public class AuthorAttribute : System.Attribute
{
    private string name;
    public double version;

    public AuthorAttribute(string name)
    {
        this.name = name;
        version = 1.0;
    }
}
```

The class name [AuthorAttribute](#) is the attribute's name, [Author](#), plus the [Attribute](#) suffix. It is derived from [System.Attribute](#), so it is a custom attribute class. The constructor's parameters are the custom attribute's positional parameters. In this example, [name](#) is a positional parameter. Any public read-write fields or properties are named parameters. In this case, [version](#) is the only named parameter. Note the use of the [AttributeUsage](#) attribute to make the [Author](#) attribute valid only on class and [struct](#) declarations.

You could use this new attribute as follows:

```
[Author("P. Ackerman", version = 1.1)]
class SampleClass
{
    // P. Ackerman's code goes here...
}
```

[AttributeUsage](#) has a named parameter, [AllowMultiple](#), with which you can make a custom attribute single-use or multiuse. In the following code example, a multiuse attribute is created.

```
[System.AttributeUsage(System.AttributeTargets.Class |
                      System.AttributeTargets.Struct,
                      AllowMultiple = true) // multiuse attribute
]
public class AuthorAttribute : System.Attribute
```

In the following code example, multiple attributes of the same type are applied to a class.

```
[Author("P. Ackerman", version = 1.1)]  
[Author("R. Koch", version = 1.2)]  
class SampleClass  
{  
    // P. Ackerman's code goes here...  
    // R. Koch's code goes here...  
}
```

See also

- [System.Reflection](#)
- [C# Programming Guide](#)
- [Writing Custom Attributes](#)
- [Reflection \(C#\)](#)
- [Attributes \(C#\)](#)
- [Accessing Attributes by Using Reflection \(C#\)](#)
- [AttributeUsage \(C#\)](#)

Accessing Attributes by Using Reflection (C#)

12/28/2021 • 2 minutes to read • [Edit Online](#)

The fact that you can define custom attributes and place them in your source code would be of little value without some way of retrieving that information and acting on it. By using reflection, you can retrieve the information that was defined with custom attributes. The key method is `GetCustomAttributes`, which returns an array of objects that are the run-time equivalents of the source code attributes. This method has several overloaded versions. For more information, see [Attribute](#).

An attribute specification such as:

```
[Author("P. Ackerman", version = 1.1)]  
class SampleClass
```

is conceptually equivalent to this:

```
Author anonymousAuthorObject = new Author("P. Ackerman");  
anonymousAuthorObject.version = 1.1;
```

However, the code is not executed until `SampleClass` is queried for attributes. Calling `GetCustomAttributes` on `SampleClass` causes an `Author` object to be constructed and initialized as above. If the class has other attributes, other attribute objects are constructed similarly. `GetCustomAttributes` then returns the `Author` object and any other attribute objects in an array. You can then iterate over this array, determine what attributes were applied based on the type of each array element, and extract information from the attribute objects.

Example

Here is a complete example. A custom attribute is defined, applied to several entities, and retrieved via reflection.

```
// Multiuse attribute.  
[System.AttributeUsage(System.AttributeTargets.Class |  
                        System.AttributeTargets.Struct,  
                        AllowMultiple = true) // Multiuse attribute.  
]  
public class Author : System.Attribute  
{  
    string name;  
    public double version;  
  
    public Author(string name)  
    {  
        this.name = name;  
  
        // Default value.  
        version = 1.0;  
    }  
  
    public string GetName()  
    {  
        return name;  
    }  
}  
  
// Class with the Author attribute.  
[Author("P. Ackerman")]
```



```

public class FirstClass
{
    // ...
}

// Class without the Author attribute.
public class SecondClass
{
    // ...
}

// Class with multiple Author attributes.
[Author("P. Ackerman"), Author("R. Koch", version = 2.0)]
public class ThirdClass
{
    // ...
}

class TestAuthorAttribute
{
    static void Test()
    {
        PrintAuthorInfo(typeof(FirstClass));
        PrintAuthorInfo(typeof(SecondClass));
        PrintAuthorInfo(typeof(ThirdClass));
    }

    private static void PrintAuthorInfo(System.Type t)
    {
        System.Console.WriteLine("Author information for {0}", t);

        // Using reflection.
        System.Attribute[] attrs = System.Attribute.GetCustomAttributes(t); // Reflection.

        // Displaying output.
        foreach (System.Attribute attr in attrs)
        {
            if (attr is Author)
            {
                Author a = (Author)attr;
                System.Console.WriteLine("    {0}, version {1:f}", a.GetName(), a.version);
            }
        }
    }
}

/* Output:
    Author information for FirstClass
        P. Ackerman, version 1.00
    Author information for SecondClass
    Author information for ThirdClass
        R. Koch, version 2.00
        P. Ackerman, version 1.00
*/

```

See also

- [System.Reflection](#)
- [Attribute](#)
- [C# Programming Guide](#)
- [Retrieving Information Stored in Attributes](#)
- [Reflection \(C#\)](#)
- [Attributes \(C#\)](#)
- [Creating Custom Attributes \(C#\)](#)

How to create a C/C++ union by using attributes (C#)

12/28/2021 • 2 minutes to read • [Edit Online](#)

By using attributes, you can customize how structs are laid out in memory. For example, you can create what is known as a union in C/C++ by using the `StructLayout(LayoutKind.Explicit)` and `FieldOffset` attributes.

Examples

In this code segment, all of the fields of `TestUnion` start at the same location in memory.

```
// Add a using directive for System.Runtime.InteropServices.

[System.Runtime.InteropServices.StructLayout(LayoutKind.Explicit)]
struct TestUnion
{
    [System.Runtime.InteropServices.FieldOffset(0)]
    public int i;

    [System.Runtime.InteropServices.FieldOffset(0)]
    public double d;

    [System.Runtime.InteropServices.FieldOffset(0)]
    public char c;

    [System.Runtime.InteropServices.FieldOffset(0)]
    public byte b;
}
```

The following is another example where fields start at different explicitly set locations.

```
// Add a using directive for System.Runtime.InteropServices.

[System.Runtime.InteropServices.StructLayout(LayoutKind.Explicit)]
struct TestExplicit
{
    [System.Runtime.InteropServices.FieldOffset(0)]
    public long lg;

    [System.Runtime.InteropServices.FieldOffset(0)]
    public int i1;

    [System.Runtime.InteropServices.FieldOffset(0)]
    public int i2;

    [System.Runtime.InteropServices.FieldOffset(8)]
    public double d;

    [System.Runtime.InteropServices.FieldOffset(12)]
    public char c;

    [System.Runtime.InteropServices.FieldOffset(14)]
    public byte b;
}
```

The two integer fields, `i1` and `i2`, share the same memory locations as `lg`. This sort of control over struct

layout is useful when using platform invocation.

See also

- [System.Reflection](#)
- [Attribute](#)
- [C# Programming Guide](#)
- [Attributes](#)
- [Reflection \(C#\)](#)
- [Attributes \(C#\)](#)
- [Creating Custom Attributes \(C#\)](#)
- [Accessing Attributes by Using Reflection \(C#\)](#)

Collections (C#)

12/28/2021 • 13 minutes to read • [Edit Online](#)

For many applications, you want to create and manage groups of related objects. There are two ways to group objects: by creating arrays of objects, and by creating collections of objects.

Arrays are most useful for creating and working with a fixed number of strongly typed objects. For information about arrays, see [Arrays](#).

Collections provide a more flexible way to work with groups of objects. Unlike arrays, the group of objects you work with can grow and shrink dynamically as the needs of the application change. For some collections, you can assign a key to any object that you put into the collection so that you can quickly retrieve the object by using the key.

A collection is a class, so you must declare an instance of the class before you can add elements to that collection.

If your collection contains elements of only one data type, you can use one of the classes in the [System.Collections.Generic](#) namespace. A generic collection enforces type safety so that no other data type can be added to it. When you retrieve an element from a generic collection, you do not have to determine its data type or convert it.

NOTE

For the examples in this topic, include [using](#) directives for the `System.Collections.Generic` and `System.Linq` namespaces.

In this topic

- [Using a Simple Collection](#)
- [Kinds of Collections](#)
 - [System.Collections.Generic Classes](#)
 - [System.Collections.Concurrent Classes](#)
 - [System.Collections Classes](#)
- [Implementing a Collection of Key/Value Pairs](#)
- [Using LINQ to Access a Collection](#)
- [Sorting a Collection](#)
- [Defining a Custom Collection](#)
- [Iterators](#)

Using a Simple Collection

The examples in this section use the generic [List<T>](#) class, which enables you to work with a strongly typed list of objects.

The following example creates a list of strings and then iterates through the strings by using a [foreach](#) statement.

```
// Create a list of strings.
var salmons = new List<string>();
salmons.Add("chinook");
salmons.Add("coho");
salmons.Add("pink");
salmons.Add("sockeye");

// Iterate through the list.
foreach (var salmon in salmons)
{
    Console.Write(salmon + " ");
}
// Output: chinook coho pink sockeye
```

If the contents of a collection are known in advance, you can use a *collection initializer* to initialize the collection. For more information, see [Object and Collection Initializers](#).

The following example is the same as the previous example, except a collection initializer is used to add elements to the collection.

```
// Create a list of strings by using a
// collection initializer.
var salmons = new List<string> { "chinook", "coho", "pink", "sockeye" };

// Iterate through the list.
foreach (var salmon in salmons)
{
    Console.Write(salmon + " ");
}
// Output: chinook coho pink sockeye
```

You can use a [for](#) statement instead of a `foreach` statement to iterate through a collection. You accomplish this by accessing the collection elements by the index position. The index of the elements starts at 0 and ends at the element count minus 1.

The following example iterates through the elements of a collection by using `for` instead of `foreach`.

```
// Create a list of strings by using a
// collection initializer.
var salmons = new List<string> { "chinook", "coho", "pink", "sockeye" };

for (var index = 0; index < salmons.Count; index++)
{
    Console.Write(salmons[index] + " ");
}
// Output: chinook coho pink sockeye
```

The following example removes an element from the collection by specifying the object to remove.

```
// Create a list of strings by using a
// collection initializer.
var salmons = new List<string> { "chinook", "coho", "pink", "sockeye" };

// Remove an element from the list by specifying
// the object.
salmons.Remove("coho");

// Iterate through the list.
foreach (var salmon in salmons)
{
    Console.Write(salmon + " ");
}
// Output: chinook pink sockeye
```

The following example removes elements from a generic list. Instead of a `foreach` statement, a `for` statement that iterates in descending order is used. This is because the [RemoveAt](#) method causes elements after a removed element to have a lower index value.

```
var numbers = new List<int> { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };

// Remove odd numbers.
for (var index = numbers.Count - 1; index >= 0; index--)
{
    if (numbers[index] % 2 == 1)
    {
        // Remove the element by specifying
        // the zero-based index in the list.
        numbers.RemoveAt(index);
    }
}

// Iterate through the list.
// A lambda expression is placed in the ForEach method
// of the List(T) object.
numbers.ForEach(
    number => Console.Write(number + " "));
// Output: 0 2 4 6 8
```

For the type of elements in the [List<T>](#), you can also define your own class. In the following example, the `Galaxy` class that is used by the [List<T>](#) is defined in the code.

```

private static void IterateThroughList()
{
    var theGalaxies = new List<Galaxy>
    {
        new Galaxy() { Name="Tadpole", MegaLightYears=400},
        new Galaxy() { Name="Pinwheel", MegaLightYears=25},
        new Galaxy() { Name="Milky Way", MegaLightYears=0},
        new Galaxy() { Name="Andromeda", MegaLightYears=3}
    };

    foreach (Galaxy theGalaxy in theGalaxies)
    {
        Console.WriteLine(theGalaxy.Name + " " + theGalaxy.MegaLightYears);
    }

    // Output:
    // Tadpole 400
    // Pinwheel 25
    // Milky Way 0
    // Andromeda 3
}

public class Galaxy
{
    public string Name { get; set; }
    public int MegaLightYears { get; set; }
}

```

Kinds of Collections

Many common collections are provided by .NET. Each type of collection is designed for a specific purpose.

Some of the common collection classes are described in this section:

- [System.Collections.Generic](#) classes
- [System.Collections.Concurrent](#) classes
- [System.Collections](#) classes

System.Collections.Generic Classes

You can create a generic collection by using one of the classes in the [System.Collections.Generic](#) namespace. A generic collection is useful when every item in the collection has the same data type. A generic collection enforces strong typing by allowing only the desired data type to be added.

The following table lists some of the frequently used classes of the [System.Collections.Generic](#) namespace:

CLASS	DESCRIPTION
Dictionary<TKey,TValue>	Represents a collection of key/value pairs that are organized based on the key.
List<T>	Represents a list of objects that can be accessed by index. Provides methods to search, sort, and modify lists.
Queue<T>	Represents a first in, first out (FIFO) collection of objects.
SortedList<TKey,TValue>	Represents a collection of key/value pairs that are sorted by key based on the associated IComparer<T> implementation.

CLASS	DESCRIPTION
Stack<T>	Represents a last in, first out (LIFO) collection of objects.

For additional information, see [Commonly Used Collection Types](#), [Selecting a Collection Class](#), and [System.Collections.Generic](#).

System.Collections.Concurrent Classes

In .NET Framework 4 and later versions, the collections in the [System.Collections.Concurrent](#) namespace provide efficient thread-safe operations for accessing collection items from multiple threads.

The classes in the [System.Collections.Concurrent](#) namespace should be used instead of the corresponding types in the [System.Collections.Generic](#) and [System.Collections](#) namespaces whenever multiple threads are accessing the collection concurrently. For more information, see [Thread-Safe Collections](#) and [System.Collections.Concurrent](#).

Some classes included in the [System.Collections.Concurrent](#) namespace are [BlockingCollection<T>](#), [ConcurrentDictionary<TKey,TValue>](#), [ConcurrentQueue<T>](#), and [ConcurrentStack<T>](#).

System.Collections Classes

The classes in the [System.Collections](#) namespace do not store elements as specifically typed objects, but as objects of type `Object`.

Whenever possible, you should use the generic collections in the [System.Collections.Generic](#) namespace or the [System.Collections.Concurrent](#) namespace instead of the legacy types in the `System.Collections` namespace.

The following table lists some of the frequently used classes in the `System.Collections` namespace:

CLASS	DESCRIPTION
ArrayList	Represents an array of objects whose size is dynamically increased as required.
Hashtable	Represents a collection of key/value pairs that are organized based on the hash code of the key.
Queue	Represents a first in, first out (FIFO) collection of objects.
Stack	Represents a last in, first out (LIFO) collection of objects.

The [System.Collections.Specialized](#) namespace provides specialized and strongly typed collection classes, such as string-only collections and linked-list and hybrid dictionaries.

Implementing a Collection of Key/Value Pairs

The [Dictionary<TKey,TValue>](#) generic collection enables you to access to elements in a collection by using the key of each element. Each addition to the dictionary consists of a value and its associated key. Retrieving a value by using its key is fast because the `Dictionary` class is implemented as a hash table.

The following example creates a `Dictionary` collection and iterates through the dictionary by using a `foreach` statement.


```

private static void IterateThruDictionary()
{
    Dictionary<string, Element> elements = BuildDictionary();

    foreach (KeyValuePair<string, Element> kvp in elements)
    {
        Element theElement = kvp.Value;

        Console.WriteLine("key: " + kvp.Key);
        Console.WriteLine("values: " + theElement.Symbol + " " +
            theElement.Name + " " + theElement.AtomicNumber);
    }
}

private static Dictionary<string, Element> BuildDictionary()
{
    var elements = new Dictionary<string, Element>();

    AddToDictionary(elements, "K", "Potassium", 19);
    AddToDictionary(elements, "Ca", "Calcium", 20);
    AddToDictionary(elements, "Sc", "Scandium", 21);
    AddToDictionary(elements, "Ti", "Titanium", 22);

    return elements;
}

private static void AddToDictionary(Dictionary<string, Element> elements,
    string symbol, string name, int atomicNumber)
{
    Element theElement = new Element();

    theElement.Symbol = symbol;
    theElement.Name = name;
    theElement.AtomicNumber = atomicNumber;

    elements.Add(key: theElement.Symbol, value: theElement);
}

public class Element
{
    public string Symbol { get; set; }
    public string Name { get; set; }
    public int AtomicNumber { get; set; }
}

```

To instead use a collection initializer to build the `Dictionary` collection, you can replace the `BuildDictionary` and `AddToDictionary` methods with the following method.

```

private static Dictionary<string, Element> BuildDictionary2()
{
    return new Dictionary<string, Element>
    {
        {"K",
            new Element() { Symbol="K", Name="Potassium", AtomicNumber=19}},
        {"Ca",
            new Element() { Symbol="Ca", Name="Calcium", AtomicNumber=20}},
        {"Sc",
            new Element() { Symbol="Sc", Name="Scandium", AtomicNumber=21}},
        {"Ti",
            new Element() { Symbol="Ti", Name="Titanium", AtomicNumber=22}}
    };
}

```

The following example uses the `ContainsKey` method and the `Item[]` property of `Dictionary` to quickly find an

item by key. The `Item` property enables you to access an item in the `elements` collection by using the `elements[symbol]` in C#.

```
private static void FindInDictionary(string symbol)
{
    Dictionary<string, Element> elements = BuildDictionary();

    if (elements.ContainsKey(symbol) == false)
    {
        Console.WriteLine(symbol + " not found");
    }
    else
    {
        Element theElement = elements[symbol];
        Console.WriteLine("found: " + theElement.Name);
    }
}
```

The following example instead uses the [TryGetValue](#) method quickly find an item by key.

```
private static void FindInDictionary2(string symbol)
{
    Dictionary<string, Element> elements = BuildDictionary();

    Element theElement = null;
    if (elements.TryGetValue(symbol, out theElement) == false)
        Console.WriteLine(symbol + " not found");
    else
        Console.WriteLine("found: " + theElement.Name);
}
```

Using LINQ to Access a Collection

LINQ (Language-Integrated Query) can be used to access collections. LINQ queries provide filtering, ordering, and grouping capabilities. For more information, see [Getting Started with LINQ in C#](#).

The following example runs a LINQ query against a generic `List`. The LINQ query returns a different collection that contains the results.

```

private static void ShowLINQ()
{
    List<Element> elements = BuildList();

    // LINQ Query.
    var subset = from theElement in elements
                 where theElement.AtomicNumber < 22
                 orderby theElement.Name
                 select theElement;

    foreach (Element theElement in subset)
    {
        Console.WriteLine(theElement.Name + " " + theElement.AtomicNumber);
    }

    // Output:
    // Calcium 20
    // Potassium 19
    // Scandium 21
}

private static List<Element> BuildList()
{
    return new List<Element>
    {
        { new Element() { Symbol="K", Name="Potassium", AtomicNumber=19}},
        { new Element() { Symbol="Ca", Name="Calcium", AtomicNumber=20}},
        { new Element() { Symbol="Sc", Name="Scandium", AtomicNumber=21}},
        { new Element() { Symbol="Ti", Name="Titanium", AtomicNumber=22}}
    };
}

public class Element
{
    public string Symbol { get; set; }
    public string Name { get; set; }
    public int AtomicNumber { get; set; }
}

```

Sorting a Collection

The following example illustrates a procedure for sorting a collection. The example sorts instances of the `Car` class that are stored in a `List<T>`. The `Car` class implements the `IComparable<T>` interface, which requires that the `CompareTo` method be implemented.

Each call to the `CompareTo` method makes a single comparison that is used for sorting. User-written code in the `CompareTo` method returns a value for each comparison of the current object with another object. The value returned is less than zero if the current object is less than the other object, greater than zero if the current object is greater than the other object, and zero if they are equal. This enables you to define in code the criteria for greater than, less than, and equal.

In the `ListCars` method, the `cars.Sort()` statement sorts the list. This call to the `Sort` method of the `List<T>` causes the `CompareTo` method to be called automatically for the `Car` objects in the `List`.

```

private static void ListCars()
{
    var cars = new List<Car>
    {
        { new Car() { Name = "car1", Color = "blue", Speed = 20}},
        { new Car() { Name = "car2", Color = "red", Speed = 50}},
        { new Car() { Name = "car3", Color = "green", Speed = 10}},
        { new Car() { Name = "car4", Color = "blue", Speed = 50}},
    }
}

```

```

        { new Car() { Name = "car5", Color = "blue", Speed = 30}},
        { new Car() { Name = "car6", Color = "red", Speed = 60}},
        { new Car() { Name = "car7", Color = "green", Speed = 50}}
    };

    // Sort the cars by color alphabetically, and then by speed
    // in descending order.
    cars.Sort();

    // View all of the cars.
    foreach (Car thisCar in cars)
    {
        Console.Write(thisCar.Color.PadRight(5) + " ");
        Console.Write(thisCar.Speed.ToString() + " ");
        Console.Write(thisCar.Name);
        Console.WriteLine();
    }

    // Output:
    // blue  50 car4
    // blue  30 car5
    // blue  20 car1
    // green 50 car7
    // green 10 car3
    // red   60 car6
    // red   50 car2
}

public class Car : IComparable<Car>
{
    public string Name { get; set; }
    public int Speed { get; set; }
    public string Color { get; set; }

    public int CompareTo(Car other)
    {
        // A call to this method makes a single comparison that is
        // used for sorting.

        // Determine the relative order of the objects being compared.
        // Sort by color alphabetically, and then by speed in
        // descending order.

        // Compare the colors.
        int compare;
        compare = String.Compare(this.Color, other.Color, true);

        // If the colors are the same, compare the speeds.
        if (compare == 0)
        {
            compare = this.Speed.CompareTo(other.Speed);

            // Use descending order for speed.
            compare = -compare;
        }

        return compare;
    }
}

```

Defining a Custom Collection

You can define a collection by implementing the [IEnumerable<T>](#) or [IEnumerable](#) interface.

Although you can define a custom collection, it is usually better to instead use the collections that are included in

.NET, which are described in [Kinds of Collections](#) earlier in this article.

The following example defines a custom collection class named `AllColors`. This class implements the [IEnumerable](#) interface, which requires that the [GetEnumerator](#) method be implemented.

The `GetEnumerator` method returns an instance of the `ColorEnumerator` class. `ColorEnumerator` implements the [IEnumerator](#) interface, which requires that the [Current](#) property, [MoveNext](#) method, and [Reset](#) method be implemented.

```
private static void ListColors()
{
    var colors = new AllColors();

    foreach (Color theColor in colors)
    {
        Console.Write(theColor.Name + " ");
    }
    Console.WriteLine();
    // Output: red blue green
}

// Collection class.
public class AllColors : System.Collections.IEnumerable
{
    Color[] _colors =
    {
        new Color() { Name = "red" },
        new Color() { Name = "blue" },
        new Color() { Name = "green" }
    };

    public System.Collections.IEnumerator GetEnumerator()
    {
        return new ColorEnumerator(_colors);

        // Instead of creating a custom enumerator, you could
        // use the GetEnumerator of the array.
        //return _colors.GetEnumerator();
    }

    // Custom enumerator.
    private class ColorEnumerator : System.Collections.IEnumerator
    {
        private Color[] _colors;
        private int _position = -1;

        public ColorEnumerator(Color[] colors)
        {
            _colors = colors;
        }

        object System.Collections.IEnumerator.Current
        {
            get
            {
                return _colors[_position];
            }
        }

        bool System.Collections.IEnumerator.MoveNext()
        {
            _position++;
            return (_position < _colors.Length);
        }

        void System.Collections.IEnumerator.Reset()
        {
            _position = -1;
        }
    }
}
```

```

        _position = -1;
    }
}

// Element class.
public class Color
{
    public string Name { get; set; }
}

```

Iterators

An *iterator* is used to perform a custom iteration over a collection. An iterator can be a method or a `get` accessor. An iterator uses a `yield return` statement to return each element of the collection one at a time.

You call an iterator by using a `foreach` statement. Each iteration of the `foreach` loop calls the iterator. When a `yield return` statement is reached in the iterator, an expression is returned, and the current location in code is retained. Execution is restarted from that location the next time that the iterator is called.

For more information, see [Iterators \(C#\)](#).

The following example uses an iterator method. The iterator method has a `yield return` statement that is inside a `for` loop. In the `ListEvenNumbers` method, each iteration of the `foreach` statement body creates a call to the iterator method, which proceeds to the next `yield return` statement.

```

private static void ListEvenNumbers()
{
    foreach (int number in EvenSequence(5, 18))
    {
        Console.Write(number.ToString() + " ");
    }
    Console.WriteLine();
    // Output: 6 8 10 12 14 16 18
}

private static IEnumerable<int> EvenSequence(
    int firstNumber, int lastNumber)
{
    // Yield even numbers in the range.
    for (var number = firstNumber; number <= lastNumber; number++)
    {
        if (number % 2 == 0)
        {
            yield return number;
        }
    }
}

```

See also

- [Object and Collection Initializers](#)
- [Programming Concepts \(C#\)](#)
- [Option Strict Statement](#)
- [LINQ to Objects \(C#\)](#)
- [Parallel LINQ \(PLINQ\)](#)
- [Collections and Data Structures](#)
- [Selecting a Collection Class](#)

- [Comparisons and Sorts Within Collections](#)
- [When to Use Generic Collections](#)
- [Iteration statements](#)

Covariance and Contravariance (C#)

12/28/2021 • 3 minutes to read • [Edit Online](#)

In C#, covariance and contravariance enable implicit reference conversion for array types, delegate types, and generic type arguments. Covariance preserves assignment compatibility and contravariance reverses it.

The following code demonstrates the difference between assignment compatibility, covariance, and contravariance.

```
// Assignment compatibility.
string str = "test";
// An object of a more derived type is assigned to an object of a less derived type.
object obj = str;

// Covariance.
IEnumerable<string> strings = new List<string>();
// An object that is instantiated with a more derived type argument
// is assigned to an object instantiated with a less derived type argument.
// Assignment compatibility is preserved.
IEnumerable<object> objects = strings;

// Contravariance.
// Assume that the following method is in the class:
// static void SetObject(object o) { }
Action<object> actObject = SetObject;
// An object that is instantiated with a less derived type argument
// is assigned to an object instantiated with a more derived type argument.
// Assignment compatibility is reversed.
Action<string> actString = actObject;
```

Covariance for arrays enables implicit conversion of an array of a more derived type to an array of a less derived type. But this operation is not type safe, as shown in the following code example.

```
object[] array = new String[10];
// The following statement produces a run-time exception.
// array[0] = 10;
```

Covariance and contravariance support for method groups allows for matching method signatures with delegate types. This enables you to assign to delegates not only methods that have matching signatures, but also methods that return more derived types (covariance) or that accept parameters that have less derived types (contravariance) than that specified by the delegate type. For more information, see [Variance in Delegates \(C#\)](#) and [Using Variance in Delegates \(C#\)](#).

The following code example shows covariance and contravariance support for method groups.


```

static object GetObject() { return null; }
static void SetObject(object obj) { }

static string GetString() { return ""; }
static void SetString(string str) { }

static void Test()
{
    // Covariance. A delegate specifies a return type as object,
    // but you can assign a method that returns a string.
    Func<object> del = GetString;

    // Contravariance. A delegate specifies a parameter type as string,
    // but you can assign a method that takes an object.
    Action<string> del2 = SetObject;
}

```

In .NET Framework 4 and later versions, C# supports covariance and contravariance in generic interfaces and delegates and allows for implicit conversion of generic type parameters. For more information, see [Variance in Generic Interfaces \(C#\)](#) and [Variance in Delegates \(C#\)](#).

The following code example shows implicit reference conversion for generic interfaces.

```

IEnumerable<String> strings = new List<String>();
IEnumerable<Object> objects = strings;

```

A generic interface or delegate is called *variant* if its generic parameters are declared covariant or contravariant. C# enables you to create your own variant interfaces and delegates. For more information, see [Creating Variant Generic Interfaces \(C#\)](#) and [Variance in Delegates \(C#\)](#).

Related Topics

TITLE	DESCRIPTION
Variance in Generic Interfaces (C#)	Discusses covariance and contravariance in generic interfaces and provides a list of variant generic interfaces in .NET.
Creating Variant Generic Interfaces (C#)	Shows how to create custom variant interfaces.
Using Variance in Interfaces for Generic Collections (C#)	Shows how covariance and contravariance support in the IEnumerable<T> and IComparable<T> interfaces can help you reuse code.
Variance in Delegates (C#)	Discusses covariance and contravariance in generic and non-generic delegates and provides a list of variant generic delegates in .NET.
Using Variance in Delegates (C#)	Shows how to use covariance and contravariance support in non-generic delegates to match method signatures with delegate types.
Using Variance for Func and Action Generic Delegates (C#)	Shows how covariance and contravariance support in the <code>Func</code> and <code>Action</code> delegates can help you reuse code.

Variance in Generic Interfaces (C#)

12/28/2021 • 2 minutes to read • [Edit Online](#)

.NET Framework 4 introduced variance support for several existing generic interfaces. Variance support enables implicit conversion of classes that implement these interfaces.

Starting with .NET Framework 4, the following interfaces are variant:

- [IEnumerable<T>](#) (T is covariant)
- [IEnumerator<T>](#) (T is covariant)
- [IQueryable<T>](#) (T is covariant)
- [IGrouping<TKey,TElement>](#) (`TKey` and `TElement` are covariant)
- [IComparer<T>](#) (T is contravariant)
- [IEqualityComparer<T>](#) (T is contravariant)
- [IComparable<T>](#) (T is contravariant)

Starting with .NET Framework 4.5, the following interfaces are variant:

- [IReadOnlyList<T>](#) (T is covariant)
- [IReadOnlyCollection<T>](#) (T is covariant)

Covariance permits a method to have a more derived return type than that defined by the generic type parameter of the interface. To illustrate the covariance feature, consider these generic interfaces:

`IEnumerable<Object>` and `IEnumerable<String>`. The `IEnumerable<String>` interface does not inherit the `IEnumerable<Object>` interface. However, the `String` type does inherit the `Object` type, and in some cases you may want to assign objects of these interfaces to each other. This is shown in the following code example.

```
IEnumerable<String> strings = new List<String>();
IEnumerable<Object> objects = strings;
```

In earlier versions of .NET Framework, this code causes a compilation error in C# and, if `Option Strict` is on, in Visual Basic. But now you can use `strings` instead of `objects`, as shown in the previous example, because the [IEnumerable<T>](#) interface is covariant.

Contravariance permits a method to have argument types that are less derived than that specified by the generic parameter of the interface. To illustrate contravariance, assume that you have created a `BaseComparer` class to compare instances of the `BaseClass` class. The `BaseComparer` class implements the `IEqualityComparer<BaseClass>` interface. Because the [IEqualityComparer<T>](#) interface is now contravariant, you can use `BaseComparer` to compare instances of classes that inherit the `BaseClass` class. This is shown in the following code example.

```
// Simple hierarchy of classes.
class BaseClass { }
class DerivedClass : BaseClass { }

// Comparer class.
class BaseComparer : IEqualityComparer<BaseClass>
{
    public int GetHashCode(BaseClass baseInstance)
    {
        return baseInstance.GetHashCode();
    }
    public bool Equals(BaseClass x, BaseClass y)
    {
        return x == y;
    }
}
class Program
{
    static void Test()
    {
        IEqualityComparer<BaseClass> baseComparer = new BaseComparer();

        // Implicit conversion of IEqualityComparer<BaseClass> to
        // IEqualityComparer<DerivedClass>.
        IEqualityComparer<DerivedClass> childComparer = baseComparer;
    }
}
```

For more examples, see [Using Variance in Interfaces for Generic Collections \(C#\)](#).

Variance in generic interfaces is supported for reference types only. Value types do not support variance. For example, `IEnumerable<int>` cannot be implicitly converted to `IEnumerable<object>`, because integers are represented by a value type.

```
IEnumerable<int> integers = new List<int>();
// The following statement generates a compiler error,
// because int is a value type.
// IEnumerable<Object> objects = integers;
```

It is also important to remember that classes that implement variant interfaces are still invariant. For example, although `List<T>` implements the covariant interface `IEnumerable<T>`, you cannot implicitly convert `List<String>` to `List<Object>`. This is illustrated in the following code example.

```
// The following line generates a compiler error
// because classes are invariant.
// List<Object> list = new List<String>();

// You can use the interface object instead.
IEnumerable<Object> listObjects = new List<String>();
```

See also

- [Using Variance in Interfaces for Generic Collections \(C#\)](#)
- [Creating Variant Generic Interfaces \(C#\)](#)
- [Generic Interfaces](#)
- [Variance in Delegates \(C#\)](#)

Creating Variant Generic Interfaces (C#)

12/28/2021 • 4 minutes to read • [Edit Online](#)

You can declare generic type parameters in interfaces as covariant or contravariant. *Covariance* allows interface methods to have more derived return types than that defined by the generic type parameters. *Contravariance* allows interface methods to have argument types that are less derived than that specified by the generic parameters. A generic interface that has covariant or contravariant generic type parameters is called *variant*.

NOTE

.NET Framework 4 introduced variance support for several existing generic interfaces. For the list of the variant interfaces in .NET, see [Variance in Generic Interfaces \(C#\)](#).

Declaring Variant Generic Interfaces

You can declare variant generic interfaces by using the `in` and `out` keywords for generic type parameters.

IMPORTANT

`ref`, `in`, and `out` parameters in C# cannot be variant. Value types also do not support variance.

You can declare a generic type parameter covariant by using the `out` keyword. The covariant type must satisfy the following conditions:

- The type is used only as a return type of interface methods and not used as a type of method arguments. This is illustrated in the following example, in which the type `R` is declared covariant.

```
interface ICovariant<out R>
{
    R GetSomething();
    // The following statement generates a compiler error.
    // void SetSomething(R sampleArg);
}
```

There is one exception to this rule. If you have a contravariant generic delegate as a method parameter, you can use the type as a generic type parameter for the delegate. This is illustrated by the type `R` in the following example. For more information, see [Variance in Delegates \(C#\)](#) and [Using Variance for Func and Action Generic Delegates \(C#\)](#).

```
interface ICovariant<out R>
{
    void DoSomething(Action<R> callback);
}
```

- The type is not used as a generic constraint for the interface methods. This is illustrated in the following code.

```
interface ICovariant<out R>
{
    // The following statement generates a compiler error
    // because you can use only contravariant or invariant types
    // in generic constraints.
    // void DoSomething<T>() where T : R;
}
```

You can declare a generic type parameter contravariant by using the `in` keyword. The contravariant type can be used only as a type of method arguments and not as a return type of interface methods. The contravariant type can also be used for generic constraints. The following code shows how to declare a contravariant interface and use a generic constraint for one of its methods.

```
interface IContravariant<in A>
{
    void SetSomething(A sampleArg);
    void DoSomething<T>() where T : A;
    // The following statement generates a compiler error.
    // A GetSomething();
}
```

It is also possible to support both covariance and contravariance in the same interface, but for different type parameters, as shown in the following code example.

```
interface IVariant<out R, in A>
{
    R GetSomething();
    void SetSomething(A sampleArg);
    R GetSetSomethings(A sampleArg);
}
```

Implementing Variant Generic Interfaces

You implement variant generic interfaces in classes by using the same syntax that is used for invariant interfaces. The following code example shows how to implement a covariant interface in a generic class.

```
interface ICovariant<out R>
{
    R GetSomething();
}
class SampleImplementation<R> : ICovariant<R>
{
    public R GetSomething()
    {
        // Some code.
        return default(R);
    }
}
```

Classes that implement variant interfaces are invariant. For example, consider the following code.

```
// The interface is covariant.
ICovariant<Button> ibutton = new SampleImplementation<Button>();
ICovariant<Object> iobj = ibutton;

// The class is invariant.
SampleImplementation<Button> button = new SampleImplementation<Button>();
// The following statement generates a compiler error
// because classes are invariant.
// SampleImplementation<Object> obj = button;
```

Extending Variant Generic Interfaces

When you extend a variant generic interface, you have to use the `in` and `out` keywords to explicitly specify whether the derived interface supports variance. The compiler does not infer the variance from the interface that is being extended. For example, consider the following interfaces.

```
interface ICovariant<out T> { }
interface IInvariant<T> : ICovariant<T> { }
interface IExtCovariant<out T> : ICovariant<T> { }
```

In the `IInvariant<T>` interface, the generic type parameter `T` is invariant, whereas in `IExtCovariant<out T>` the type parameter is covariant, although both interfaces extend the same interface. The same rule is applied to contravariant generic type parameters.

You can create an interface that extends both the interface where the generic type parameter `T` is covariant and the interface where it is contravariant if in the extending interface the generic type parameter `T` is invariant. This is illustrated in the following code example.

```
interface ICovariant<out T> { }
interface IContravariant<in T> { }
interface IInvariant<T> : ICovariant<T>, IContravariant<T> { }
```

However, if a generic type parameter `T` is declared covariant in one interface, you cannot declare it contravariant in the extending interface, or vice versa. This is illustrated in the following code example.

```
interface ICovariant<out T> { }
// The following statement generates a compiler error.
// interface ICoContraVariant<in T> : ICovariant<T> { }
```

Avoiding Ambiguity

When you implement variant generic interfaces, variance can sometimes lead to ambiguity. Such ambiguity should be avoided.

For example, if you explicitly implement the same variant generic interface with different generic type parameters in one class, it can create ambiguity. The compiler does not produce an error in this case, but it's not specified which interface implementation will be chosen at run time. This ambiguity could lead to subtle bugs in your code. Consider the following code example.

```

// Simple class hierarchy.
class Animal { }
class Cat : Animal { }
class Dog : Animal { }

// This class introduces ambiguity
// because IEnumerable<out T> is covariant.
class Pets : IEnumerable<Cat>, IEnumerable<Dog>
{
    IEnumerator<Cat> IEnumerable<Cat>.GetEnumerator()
    {
        Console.WriteLine("Cat");
        // Some code.
        return null;
    }

    IEnumerator IEnumerable.GetEnumerator()
    {
        // Some code.
        return null;
    }

    IEnumerator<Dog> IEnumerable<Dog>.GetEnumerator()
    {
        Console.WriteLine("Dog");
        // Some code.
        return null;
    }
}
class Program
{
    public static void Test()
    {
        IEnumerable<Animal> pets = new Pets();
        pets.GetEnumerator();
    }
}

```

In this example, it is unspecified how the `pets.GetEnumerator` method chooses between `Cat` and `Dog`. This could cause problems in your code.

See also

- [Variance in Generic Interfaces \(C#\)](#)
- [Using Variance for Func and Action Generic Delegates \(C#\)](#)

Using Variance in Interfaces for Generic Collections (C#)

12/28/2021 • 2 minutes to read • [Edit Online](#)

A covariant interface allows its methods to return more derived types than those specified in the interface. A contravariant interface allows its methods to accept parameters of less derived types than those specified in the interface.

In .NET Framework 4, several existing interfaces became covariant and contravariant. These include [IEnumerable<T>](#) and [IComparable<T>](#). This enables you to reuse methods that operate with generic collections of base types for collections of derived types.

For a list of variant interfaces in .NET, see [Variance in Generic Interfaces \(C#\)](#).

Converting Generic Collections

The following example illustrates the benefits of covariance support in the [IEnumerable<T>](#) interface. The `PrintFullName` method accepts a collection of the `IEnumerable<Person>` type as a parameter. However, you can reuse it for a collection of the `IEnumerable<Employee>` type because `Employee` inherits `Person`.

```
// Simple hierarchy of classes.
public class Person
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
}

public class Employee : Person { }

class Program
{
    // The method has a parameter of the IEnumerable<Person> type.
    public static void PrintFullName(IEnumerable<Person> persons)
    {
        foreach (Person person in persons)
        {
            Console.WriteLine("Name: {0} {1}",
                person.FirstName, person.LastName);
        }
    }

    public static void Test()
    {
        IEnumerable<Employee> employees = new List<Employee>();

        // You can pass IEnumerable<Employee>,
        // although the method expects IEnumerable<Person>.

        PrintFullName(employees);
    }
}
```

Comparing Generic Collections

The following example illustrates the benefits of contravariance support in the [IEqualityComparer<T>](#) interface. The `PersonComparer` class implements the `IEqualityComparer<Person>` interface. However, you can reuse this class to compare a sequence of objects of the `Employee` type because `Employee` inherits `Person`.

```
// Simple hierarchy of classes.
public class Person
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
}

public class Employee : Person { }

// The custom comparer for the Person type
// with standard implementations of Equals()
// and GetHashCode() methods.
class PersonComparer : IEqualityComparer<Person>
{
    public bool Equals(Person x, Person y)
    {
        if (Object.ReferenceEquals(x, y)) return true;
        if (Object.ReferenceEquals(x, null) ||
            Object.ReferenceEquals(y, null))
            return false;
        return x.FirstName == y.FirstName && x.LastName == y.LastName;
    }
    public int GetHashCode(Person person)
    {
        if (Object.ReferenceEquals(person, null)) return 0;
        int hashFirstName = person.FirstName == null
            ? 0 : person.FirstName.GetHashCode();
        int hashLastName = person.LastName.GetHashCode();
        return hashFirstName ^ hashLastName;
    }
}

class Program
{
    public static void Test()
    {
        List<Employee> employees = new List<Employee> {
            new Employee() {FirstName = "Michael", LastName = "Alexander"},
            new Employee() {FirstName = "Jeff", LastName = "Price"}
        };

        // You can pass PersonComparer,
        // which implements IEqualityComparer<Person>,
        // although the method expects IEqualityComparer<Employee>.

        IEnumerable<Employee> noduplicates =
            employees.Distinct<Employee>(new PersonComparer());

        foreach (var employee in noduplicates)
            Console.WriteLine(employee.FirstName + " " + employee.LastName);
    }
}
```

See also

- [Variance in Generic Interfaces \(C#\)](#)

Variance in Delegates (C#)

12/28/2021 • 5 minutes to read • [Edit Online](#)

.NET Framework 3.5 introduced variance support for matching method signatures with delegate types in all delegates in C#. This means that you can assign to delegates not only methods that have matching signatures, but also methods that return more derived types (covariance) or that accept parameters that have less derived types (contravariance) than that specified by the delegate type. This includes both generic and non-generic delegates.

For example, consider the following code, which has two classes and two delegates: generic and non-generic.

```
public class First { }
public class Second : First { }
public delegate First SampleDelegate(Second a);
public delegate R SampleGenericDelegate<A, R>(A a);
```

When you create delegates of the `SampleDelegate` or `SampleGenericDelegate<A, R>` types, you can assign any one of the following methods to those delegates.

```
// Matching signature.
public static First ASecondRFirst(Second second)
{ return new First(); }

// The return type is more derived.
public static Second ASecondRSecond(Second second)
{ return new Second(); }

// The argument type is less derived.
public static First AFirstRFirst(First first)
{ return new First(); }

// The return type is more derived
// and the argument type is less derived.
public static Second AFirstRSecond(First first)
{ return new Second(); }
```

The following code example illustrates the implicit conversion between the method signature and the delegate type.

```
// Assigning a method with a matching signature
// to a non-generic delegate. No conversion is necessary.
SampleDelegate dNonGeneric = ASecondRFirst;
// Assigning a method with a more derived return type
// and less derived argument type to a non-generic delegate.
// The implicit conversion is used.
SampleDelegate dNonGenericConversion = AFirstRSecond;

// Assigning a method with a matching signature to a generic delegate.
// No conversion is necessary.
SampleGenericDelegate<Second, First> dGeneric = ASecondRFirst;
// Assigning a method with a more derived return type
// and less derived argument type to a generic delegate.
// The implicit conversion is used.
SampleGenericDelegate<Second, First> dGenericConversion = AFirstRSecond;
```

For more examples, see [Using Variance in Delegates \(C#\)](#) and [Using Variance for Func and Action Generic Delegates \(C#\)](#).

Variance in Generic Type Parameters

In .NET Framework 4 or later you can enable implicit conversion between delegates, so that generic delegates that have different types specified by generic type parameters can be assigned to each other, if the types are inherited from each other as required by variance.

To enable implicit conversion, you must explicitly declare generic parameters in a delegate as covariant or contravariant by using the `in` or `out` keyword.

The following code example shows how you can create a delegate that has a covariant generic type parameter.

```
// Type T is declared covariant by using the out keyword.
public delegate T SampleGenericDelegate <out T>();

public static void Test()
{
    SampleGenericDelegate <String> dString = () => " ";

    // You can assign delegates to each other,
    // because the type T is declared covariant.
    SampleGenericDelegate <Object> dObject = dString;
}
```

If you use only variance support to match method signatures with delegate types and do not use the `in` and `out` keywords, you may find that sometimes you can instantiate delegates with identical lambda expressions or methods, but you cannot assign one delegate to another.

In the following code example, `SampleGenericDelegate<String>` cannot be explicitly converted to `SampleGenericDelegate<Object>`, although `String` inherits `Object`. You can fix this problem by marking the generic parameter `T` with the `out` keyword.

```
public delegate T SampleGenericDelegate<T>();

public static void Test()
{
    SampleGenericDelegate<String> dString = () => " ";

    // You can assign the dObject delegate
    // to the same lambda expression as dString delegate
    // because of the variance support for
    // matching method signatures with delegate types.
    SampleGenericDelegate<Object> dObject = () => " ";

    // The following statement generates a compiler error
    // because the generic type T is not marked as covariant.
    // SampleGenericDelegate <Object> dObject = dString;
}
```

Generic Delegates That Have Variant Type Parameters in .NET

.NET Framework 4 introduced variance support for generic type parameters in several existing generic delegates:

- `Action` delegates from the [System](#) namespace, for example, `Action<T>` and `Action<T1,T2>`
- `Func` delegates from the [System](#) namespace, for example, `Func<TResult>` and `Func<T,TResult>`

- The [Predicate<T>](#) delegate
- The [Comparison<T>](#) delegate
- The [Converter<TInput,TOutput>](#) delegate

For more information and examples, see [Using Variance for Func and Action Generic Delegates \(C#\)](#).

Declaring Variant Type Parameters in Generic Delegates

If a generic delegate has covariant or contravariant generic type parameters, it can be referred to as a *variant generic delegate*.

You can declare a generic type parameter covariant in a generic delegate by using the `out` keyword. The covariant type can be used only as a method return type and not as a type of method arguments. The following code example shows how to declare a covariant generic delegate.

```
public delegate R DCovariant<out R>();
```

You can declare a generic type parameter contravariant in a generic delegate by using the `in` keyword. The contravariant type can be used only as a type of method arguments and not as a method return type. The following code example shows how to declare a contravariant generic delegate.

```
public delegate void DContravariant<in A>(A a);
```

IMPORTANT

`ref`, `in`, and `out` parameters in C# can't be marked as variant.

It is also possible to support both variance and covariance in the same delegate, but for different type parameters. This is shown in the following example.

```
public delegate R DVariant<in A, out R>(A a);
```

Instantiating and Invoking Variant Generic Delegates

You can instantiate and invoke variant delegates just as you instantiate and invoke invariant delegates. In the following example, the delegate is instantiated by a lambda expression.

```
DVariant<String, String> dvariant = (String str) => str + " ";  
dvariant("test");
```

Combining Variant Generic Delegates

Don't combine variant delegates. The [Combine](#) method does not support variant delegate conversion and expects delegates to be of exactly the same type. This can lead to a run-time exception when you combine delegates either by using the [Combine](#) method or by using the `+` operator, as shown in the following code example.

```

Action<object> actObj = x => Console.WriteLine("object: {0}", x);
Action<string> actStr = x => Console.WriteLine("string: {0}", x);
// All of the following statements throw exceptions at run time.
// Action<string> actCombine = actStr + actObj;
// actStr += actObj;
// Delegate.Combine(actStr, actObj);

```

Variance in Generic Type Parameters for Value and Reference Types

Variance for generic type parameters is supported for reference types only. For example, `DVariant<int>` can't be implicitly converted to `DVariant<Object>` or `DVariant<long>`, because integer is a value type.

The following example demonstrates that variance in generic type parameters is not supported for value types.

```

// The type T is covariant.
public delegate T DVariant<out T>();

// The type T is invariant.
public delegate T DInvariant<T>();

public static void Test()
{
    int i = 0;
    DInvariant<int> dInt = () => i;
    DVariant<int> dVariantInt = () => i;

    // All of the following statements generate a compiler error
    // because type variance in generic parameters is not supported
    // for value types, even if generic type parameters are declared variant.
    // DInvariant<Object> dObject = dInt;
    // DInvariant<long> dLong = dInt;
    // DVariant<Object> dVariantObject = dVariantInt;
    // DVariant<long> dVariantLong = dVariantInt;
}

```

See also

- [Generics](#)
- [Using Variance for Func and Action Generic Delegates \(C#\)](#)
- [How to combine delegates \(Multicast Delegates\)](#)

Using Variance in Delegates (C#)

12/28/2021 • 2 minutes to read • [Edit Online](#)

When you assign a method to a delegate, *covariance* and *contravariance* provide flexibility for matching a delegate type with a method signature. Covariance permits a method to have return type that is more derived than that defined in the delegate. Contravariance permits a method that has parameter types that are less derived than those in the delegate type.

Example 1: Covariance

Description

This example demonstrates how delegates can be used with methods that have return types that are derived from the return type in the delegate signature. The data type returned by `DogsHandler` is of type `Dogs`, which derives from the `Mammals` type that is defined in the delegate.

Code

```
class Mammals {}
class Dogs : Mammals {}

class Program
{
    // Define the delegate.
    public delegate Mammals HandlerMethod();

    public static Mammals MammalsHandler()
    {
        return null;
    }

    public static Dogs DogsHandler()
    {
        return null;
    }

    static void Test()
    {
        HandlerMethod handlerMammals = MammalsHandler;

        // Covariance enables this assignment.
        HandlerMethod handlerDogs = DogsHandler;
    }
}
```

Example 2: Contravariance

Description

This example demonstrates how delegates can be used with methods that have parameters whose types are base types of the delegate signature parameter type. With contravariance, you can use one event handler instead of separate handlers. The following example makes use of two delegates:

- A `KeyEventHandler` delegate that defines the signature of the `Button.KeyDown` event. Its signature is:

```
public delegate void KeyEventHandler(object sender, KeyEventArgs e)
```

- A [MouseEventHandler](#) delegate that defines the signature of the [Button.MouseClick](#) event. Its signature is:

```
public delegate void MouseEventHandler(object sender, MouseEventArgs e)
```

The example defines an event handler with an [EventArgs](#) parameter and uses it to handle both the `Button.KeyDown` and `Button.MouseClick` events. It can do this because [EventArgs](#) is a base type of both [KeyEventArgs](#) and [MouseEventArgs](#).

Code

```
// Event handler that accepts a parameter of the EventArgs type.
private void MultiHandler(object sender, System.EventArgs e)
{
    label1.Text = System.DateTime.Now.ToString();
}

public Form1()
{
    InitializeComponent();

    // You can use a method that has an EventArgs parameter,
    // although the event expects the KeyEventArgs parameter.
    this.button1.KeyDown += this.MultiHandler;

    // You can use the same method
    // for an event that expects the MouseEventArgs parameter.
    this.button1.MouseClick += this.MultiHandler;
}
```

See also

- [Variance in Delegates \(C#\)](#)
- [Using Variance for Func and Action Generic Delegates \(C#\)](#)

Using Variance for Func and Action Generic Delegates (C#)

12/28/2021 • 2 minutes to read • [Edit Online](#)

These examples demonstrate how to use covariance and contravariance in the `Func` and `Action` generic delegates to enable reuse of methods and provide more flexibility in your code.

For more information about covariance and contravariance, see [Variance in Delegates \(C#\)](#).

Using Delegates with Covariant Type Parameters

The following example illustrates the benefits of covariance support in the generic `Func` delegates. The `FindByTitle` method takes a parameter of the `String` type and returns an object of the `Employee` type. However, you can assign this method to the `Func<String, Person>` delegate because `Employee` inherits `Person`.

```
// Simple hierarchy of classes.
public class Person { }
public class Employee : Person { }
class Program
{
    static Employee FindByTitle(String title)
    {
        // This is a stub for a method that returns
        // an employee that has the specified title.
        return new Employee();
    }

    static void Test()
    {
        // Create an instance of the delegate without using variance.
        Func<String, Employee> findEmployee = FindByTitle;

        // The delegate expects a method to return Person,
        // but you can assign it a method that returns Employee.
        Func<String, Person> findPerson = FindByTitle;

        // You can also assign a delegate
        // that returns a more derived type
        // to a delegate that returns a less derived type.
        findPerson = findEmployee;
    }
}
```

Using Delegates with Contravariant Type Parameters

The following example illustrates the benefits of contravariance support in the generic `Action` delegates. The `AddToContacts` method takes a parameter of the `Person` type. However, you can assign this method to the `Action<Employee>` delegate because `Employee` inherits `Person`.


```

public class Person { }
public class Employee : Person { }
class Program
{
    static void AddToContacts(Person person)
    {
        // This method adds a Person object
        // to a contact list.
    }

    static void Test()
    {
        // Create an instance of the delegate without using variance.
        Action<Person> addPersonToContacts = AddToContacts;

        // The Action delegate expects
        // a method that has an Employee parameter,
        // but you can assign it a method that has a Person parameter
        // because Employee derives from Person.
        Action<Employee> addEmployeeToContacts = AddToContacts;

        // You can also assign a delegate
        // that accepts a less derived parameter to a delegate
        // that accepts a more derived parameter.
        addEmployeeToContacts = addPersonToContacts;
    }
}

```

See also

- [Covariance and Contravariance \(C#\)](#)
- [Generics](#)

Expression Trees (C#)

12/28/2021 • 4 minutes to read • [Edit Online](#)

Expression trees represent code in a tree-like data structure, where each node is an expression, for example, a method call or a binary operation such as `x < y`.

You can compile and run code represented by expression trees. This enables dynamic modification of executable code, the execution of LINQ queries in various databases, and the creation of dynamic queries. For more information about expression trees in LINQ, see [How to use expression trees to build dynamic queries \(C#\)](#).

Expression trees are also used in the dynamic language runtime (DLR) to provide interoperability between dynamic languages and .NET and to enable compiler writers to emit expression trees instead of Microsoft intermediate language (MSIL). For more information about the DLR, see [Dynamic Language Runtime Overview](#).

You can have the C# or Visual Basic compiler create an expression tree for you based on an anonymous lambda expression, or you can create expression trees manually by using the [System.Linq.Expressions](#) namespace.

Creating Expression Trees from Lambda Expressions

When a lambda expression is assigned to a variable of type [Expression<TDelegate>](#), the compiler emits code to build an expression tree that represents the lambda expression.

The C# compiler can generate expression trees only from expression lambdas (or single-line lambdas). It cannot parse statement lambdas (or multi-line lambdas). For more information about lambda expressions in C#, see [Lambda Expressions](#).

The following code examples demonstrate how to have the C# compiler create an expression tree that represents the lambda expression `num => num < 5`.

```
Expression<Func<int, bool>> lambda = num => num < 5;
```

Creating Expression Trees by Using the API

To create expression trees by using the API, use the [Expression](#) class. This class contains static factory methods that create expression tree nodes of specific types, for example, [ParameterExpression](#), which represents a variable or parameter, or [MethodCallExpression](#), which represents a method call. [ParameterExpression](#), [MethodCallExpression](#), and the other expression-specific types are also defined in the [System.Linq.Expressions](#) namespace. These types derive from the abstract type [Expression](#).

The following code example demonstrates how to create an expression tree that represents the lambda expression `num => num < 5` by using the API.

```
// Add the following using directive to your code file:
// using System.Linq.Expressions;

// Manually build the expression tree for
// the lambda expression num => num < 5.
ParameterExpression numParam = Expression.Parameter(typeof(int), "num");
ConstantExpression five = Expression.Constant(5, typeof(int));
BinaryExpression numLessThanFive = Expression.LessThan(numParam, five);
Expression<Func<int, bool>> lambda1 =
    Expression.Lambda<Func<int, bool>>(
        numLessThanFive,
        new ParameterExpression[] { numParam });
```

In .NET Framework 4 or later, the expression trees API also supports assignments and control flow expressions such as loops, conditional blocks, and `try-catch` blocks. By using the API, you can create expression trees that are more complex than those that can be created from lambda expressions by the C# compiler. The following example demonstrates how to create an expression tree that calculates the factorial of a number.

```
// Creating a parameter expression.
ParameterExpression value = Expression.Parameter(typeof(int), "value");

// Creating an expression to hold a local variable.
ParameterExpression result = Expression.Parameter(typeof(int), "result");

// Creating a label to jump to from a loop.
LabelTarget label = Expression.Label(typeof(int));

// Creating a method body.
BlockExpression block = Expression.Block(
    // Adding a local variable.
    new[] { result },
    // Assigning a constant to a local variable: result = 1
    Expression.Assign(result, Expression.Constant(1)),
    // Adding a loop.
    Expression.Loop(
        // Adding a conditional block into the loop.
        Expression.IfThenElse(
            // Condition: value > 1
            Expression.GreaterThan(value, Expression.Constant(1)),
            // If true: result *= value --
            Expression.MultiplyAssign(result,
                Expression.PostDecrementAssign(value)),
            // If false, exit the loop and go to the label.
            Expression.Break(label, result)
        ),
        // Label to jump to.
        label
    )
);

// Compile and execute an expression tree.
int factorial = Expression.Lambda<Func<int, int>>(block, value).Compile()(5);

Console.WriteLine(factorial);
// Prints 120.
```

For more information, see [Generating Dynamic Methods with Expression Trees in Visual Studio 2010](#), which also applies to later versions of Visual Studio.

Parsing Expression Trees

The following code example demonstrates how the expression tree that represents the lambda expression

`num => num < 5` can be decomposed into its parts.

```
// Add the following using directive to your code file:
// using System.Linq.Expressions;

// Create an expression tree.
Expression<Func<int, bool>> exprTree = num => num < 5;

// Decompose the expression tree.
ParameterExpression param = (ParameterExpression)exprTree.Parameters[0];
BinaryExpression operation = (BinaryExpression)exprTree.Body;
ParameterExpression left = (ParameterExpression)operation.Left;
ConstantExpression right = (ConstantExpression)operation.Right;

Console.WriteLine("Decomposed expression: {0} => {1} {2} {3}",
    param.Name, left.Name, operation.NodeType, right.Value);

// This code produces the following output:

// Decomposed expression: num => num LessThan 5
```

Immutability of Expression Trees

Expression trees should be immutable. This means that if you want to modify an expression tree, you must construct a new expression tree by copying the existing one and replacing nodes in it. You can use an expression tree visitor to traverse the existing expression tree. For more information, see [How to modify expression trees \(C#\)](#).

Compiling Expression Trees

The [Expression<TDelegate>](#) type provides the [Compile](#) method that compiles the code represented by an expression tree into an executable delegate.

The following code example demonstrates how to compile an expression tree and run the resulting code.

```
// Creating an expression tree.
Expression<Func<int, bool>> expr = num => num < 5;

// Compiling the expression tree into a delegate.
Func<int, bool> result = expr.Compile();

// Invoking the delegate and writing the result to the console.
Console.WriteLine(result(4));

// Prints True.

// You can also use simplified syntax
// to compile and run an expression tree.
// The following line can replace two previous statements.
Console.WriteLine(expr.Compile()(4));

// Also prints True.
```

For more information, see [How to execute expression trees \(C#\)](#).

See also

- [System.Linq.Expressions](#)
- [How to execute expression trees \(C#\)](#)

- [How to modify expression trees \(C#\)](#)
- [Lambda Expressions](#)
- [Dynamic Language Runtime Overview](#)
- [Programming Concepts \(C#\)](#)

How to execute expression trees (C#)

12/28/2021 • 2 minutes to read • [Edit Online](#)

This topic shows you how to execute an expression tree. Executing an expression tree may return a value, or it may just perform an action such as calling a method.

Only expression trees that represent lambda expressions can be executed. Expression trees that represent lambda expressions are of type [LambdaExpression](#) or [Expression<TDelegate>](#). To execute these expression trees, call the [Compile](#) method to create an executable delegate, and then invoke the delegate.

NOTE

If the type of the delegate is not known, that is, the lambda expression is of type [LambdaExpression](#) and not [Expression<TDelegate>](#), you must call the [DynamicInvoke](#) method on the delegate instead of invoking it directly.

If an expression tree does not represent a lambda expression, you can create a new lambda expression that has the original expression tree as its body, by calling the [Lambda<TDelegate>\(Expression, IEnumerable<ParameterExpression>\)](#) method. Then, you can execute the lambda expression as described earlier in this section.

Example

The following code example demonstrates how to execute an expression tree that represents raising a number to a power by creating a lambda expression and executing it. The result, which represents the number raised to the power, is displayed.

```
// The expression tree to execute.
BinaryExpression be = Expression.Power(Expression.Constant(2D), Expression.Constant(3D));

// Create a lambda expression.
Expression<Func<double>> le = Expression.Lambda<Func<double>>>(be);

// Compile the lambda expression.
Func<double> compiledExpression = le.Compile();

// Execute the lambda expression.
double result = compiledExpression();

// Display the result.
Console.WriteLine(result);

// This code produces the following output:
// 8
```

Compiling the Code

- Include the [System.Linq.Expressions](#) namespace.

See also

- [Expression Trees \(C#\)](#)
- [How to modify expression trees \(C#\)](#)

How to modify expression trees (C#)

12/28/2021 • 2 minutes to read • [Edit Online](#)

This topic shows you how to modify an expression tree. Expression trees are immutable, which means that they cannot be modified directly. To change an expression tree, you must create a copy of an existing expression tree and when you create the copy, make the required changes. You can use the [ExpressionVisitor](#) class to traverse an existing expression tree and to copy each node that it visits.

To modify an expression tree

1. Create a new **Console Application** project.
2. Add a `using` directive to the file for the `System.Linq.Expressions` namespace.
3. Add the `AndAlsoModifier` class to your project.

```
public class AndAlsoModifier : ExpressionVisitor
{
    public Expression Modify(Expression expression)
    {
        return Visit(expression);
    }

    protected override Expression VisitBinary(BinaryExpression b)
    {
        if (b.NodeType == ExpressionType.AndAlso)
        {
            Expression left = this.Visit(b.Left);
            Expression right = this.Visit(b.Right);

            // Make this binary expression an OrElse operation instead of an AndAlso operation.
            return Expression.MakeBinary(ExpressionType.OrElse, left, right, b.IsLiftedToNull,
b.Method);
        }

        return base.VisitBinary(b);
    }
}
```

This class inherits the [ExpressionVisitor](#) class and is specialized to modify expressions that represent conditional `AND` operations. It changes these operations from a conditional `AND` to a conditional `OR`. To do this, the class overrides the [VisitBinary](#) method of the base type, because conditional `AND` expressions are represented as binary expressions. In the `VisitBinary` method, if the expression that is passed to it represents a conditional `AND` operation, the code constructs a new expression that contains the conditional `OR` operator instead of the conditional `AND` operator. If the expression that is passed to `VisitBinary` does not represent a conditional `AND` operation, the method defers to the base class implementation. The base class methods construct nodes that are like the expression trees that are passed in, but the nodes have their sub trees replaced with the expression trees that are produced recursively by the visitor.

4. Add a `using` directive to the file for the `System.Linq.Expressions` namespace.
5. Add code to the `Main` method in the Program.cs file to create an expression tree and pass it to the method that will modify it.

```

Expression<Func<string, bool>> expr = name => name.Length > 10 && name.StartsWith("G");
Console.WriteLine(expr);

AndAlsoModifier treeModifier = new AndAlsoModifier();
Expression modifiedExpr = treeModifier.Modify((Expression) expr);

Console.WriteLine(modifiedExpr);

/* This code produces the following output:

    name => ((name.Length > 10) && name.StartsWith("G"))
    name => ((name.Length > 10) || name.StartsWith("G"))
*/

```

The code creates an expression that contains a conditional `AND` operation. It then creates an instance of the `AndAlsoModifier` class and passes the expression to the `Modify` method of this class. Both the original and the modified expression trees are outputted to show the change.

6. Compile and run the application.

See also

- [How to execute expression trees \(C#\)](#)
- [Expression Trees \(C#\)](#)

Querying based on runtime state (C#)

12/28/2021 • 8 minutes to read • [Edit Online](#)

Consider code that defines an [IQueryable](#) or an [IQueryable<T>](#) against a data source:

```
var companyNames = new[] {  
    "Consolidated Messenger", "Alpine Ski House", "Southridge Video",  
    "City Power & Light", "Coho Winery", "Wide World Importers",  
    "Graphic Design Institute", "Adventure Works", "Humongous Insurance",  
    "Woodgrove Bank", "Margie's Travel", "Northwind Traders",  
    "Blue Yonder Airlines", "Trey Research", "The Phone Company",  
    "Wingtip Toys", "Lucerne Publishing", "Fourth Coffee"  
};  
  
// We're using an in-memory array as the data source, but the IQueryable could have come  
// from anywhere -- an ORM backed by a database, a web request, or any other LINQ provider.  
IQueryable<string> companyNamesSource = companyNames.AsQueryable();  
var fixedQry = companyNames.OrderBy(x => x);
```

Every time you run this code, the same exact query will be executed. This is frequently not very useful, as you may want your code to execute different queries depending on conditions at run time. This article describes how you can execute a different query based on runtime state.

IQueryable / IQueryable<T> and expression trees

Fundamentally, an [IQueryable](#) has two components:

- [Expression](#)—a language- and datasource-agnostic representation of the current query's components, in the form of an expression tree.
- [Provider](#)—an instance of a LINQ provider, which knows how to materialize the current query into a value or set of values.

In the context of dynamic querying, the provider will usually remain the same; the expression tree of the query will differ from query to query.

Expression trees are immutable; if you want a different expression tree—and thus a different query—you'll need to translate the existing expression tree to a new one, and thus to a new [IQueryable](#).

The following sections describe specific techniques for querying differently in response to runtime state:

- Use runtime state from within the expression tree
- Call additional LINQ methods
- Vary the expression tree passed into the LINQ methods
- Construct an [Expression<TDelegate>](#) expression tree using the factory methods at [Expression](#)
- Add method call nodes to an [IQueryable](#)'s expression tree
- Construct strings, and use the [Dynamic LINQ library](#)

Use runtime state from within the expression tree

Assuming the LINQ provider supports it, the simplest way to query dynamically is to reference the runtime state directly in the query via a closed-over variable, such as `length` in the following code example:

```

var length = 1;
var qry = companyNamesSource
    .Select(x => x.Substring(0, length))
    .Distinct();

Console.WriteLine(string.Join(", ", qry));
// prints: C, A, S, W, G, H, M, N, B, T, L, F

length = 2;
Console.WriteLine(string.Join(", ", qry));
// prints: Co, Al, So, Ci, Wi, Gr, Ad, Hu, Wo, Ma, No, Bl, Tr, Th, Lu, Fo

```

The internal expression tree—and thus the query—haven't been modified; the query returns different values only because the value of `length` has been changed.

Call additional LINQ methods

Generally, the [built-in LINQ methods](#) at [Queryable](#) perform two steps:

- Wrap the current expression tree in a [MethodCallExpression](#) representing the method call.
- Pass the wrapped expression tree back to the provider, either to return a value via the provider's [IQueryProvider.Execute](#) method; or to return a translated query object via the [IQueryProvider.CreateQuery](#) method.

You can replace the original query with the result of an [IQueryable<T>](#)-returning method, to get a new query. You can do this conditionally based on runtime state, as in the following example:

```

// bool sortByLength = /* ... */;

var qry = companyNamesSource;
if (sortByLength)
{
    qry = qry.OrderBy(x => x.Length);
}

```

Vary the expression tree passed into the LINQ methods

You can pass in different expressions to the LINQ methods, depending on runtime state:

```

// string? startsWith = /* ... */;
// string? endsWith = /* ... */;

Expression<Func<string, bool>> expr = (startsWith, endsWith) switch
{
    (" " or null, " " or null) => x => true,
    (_, " " or null) => x => x.StartsWith(startsWith),
    (" " or null, _) => x => x.EndsWith(endsWith),
    (_, _) => x => x.StartsWith(startsWith) || x.EndsWith(endsWith)
};

var qry = companyNamesSource.Where(expr);

```

You might also want to compose the various subexpressions using a third-party library such as [LinqKit's PredicateBuilder](#):

```
// This is functionally equivalent to the previous example.

// using LinqKit;
// string? startsWith = /* ... */;
// string? endsWith = /* ... */;

Expression<Func<string, bool>>? expr = PredicateBuilder.New<string>(false);
var original = expr;
if (!string.IsNullOrEmpty(startsWith))
{
    expr = expr.Or(x => x.StartsWith(startsWith));
}
if (!string.IsNullOrEmpty(endsWith))
{
    expr = expr.Or(x => x.EndsWith(endsWith));
}
if (expr == original)
{
    expr = x => true;
}

var qry = companyNamesSource.Where(expr);
```

Construct expression trees and queries using factory methods

In all the examples up to this point, we've known the element type at compile time—`string`—and thus the type of the query—`IQueryable<string>`. You may need to add components to a query of any element type, or to add different components, depending on the element type. You can create expression trees from the ground up, using the factory methods at [System.Linq.Expressions.Expression](#), and thus tailor the expression at run time to a specific element type.

Constructing an `Expression<TDelegate>`

When you construct an expression to pass into one of the LINQ methods, you're actually constructing an instance of `Expression<TDelegate>`, where `TDelegate` is some delegate type such as `Func<string, bool>`, `Action`, or a custom delegate type.

`Expression<TDelegate>` inherits from `LambdaExpression`, which represents a complete lambda expression like the following:

```
Expression<Func<string, bool>> expr = x => x.StartsWith("a");
```

A `LambdaExpression` has two components:

- A parameter list—`(string x)`—represented by the `Parameters` property.
- A body—`x.StartsWith("a")`—represented by the `Body` property.

The basic steps in constructing an `Expression<TDelegate>` are as follows:

- Define `ParameterExpression` objects for each of the parameters (if any) in the lambda expression, using the `Parameter` factory method.

```
ParameterExpression x = Parameter(typeof(string), "x");
```

- Construct the body of your `LambdaExpression`, using the `ParameterExpression`(s) you've defined, and the factory methods at `Expression`. For instance, an expression representing `x.StartsWith("a")` could be constructed like this:

```
Expression body = Call(
    x,
    typeof(string).GetMethod("StartsWith", new[] { typeof(string) }!),
    Constant("a")
);
```

- Wrap the parameters and body in a compile-time-typed `Expression<TDelegate>`, using the appropriate `Lambda` factory method overload:

```
Expression<Func<string, bool>> expr = Lambda<Func<string, bool>>(body, x);
```

The following sections describe a scenario in which you might want to construct an `Expression<TDelegate>` to pass into a LINQ method, and provide a complete example of how to do so using the factory methods.

Scenario

Let's say you have multiple entity types:

```
record Person(string LastName, string FirstName, DateTime DateOfBirth);
record Car(string Model, int Year);
```

For any of these entity types, you want to filter and return only those entities that have a given text inside one of their `string` fields. For `Person`, you'd want to search the `FirstName` and `LastName` properties:

```
string term = /* ... */;
var personsQry = new List<Person>()
    .AsQueryable()
    .Where(x => x.FirstName.Contains(term) || x.LastName.Contains(term));
```

But for `car`, you'd want to search only the `Model` property:

```
string term = /* ... */;
var carsQry = new List<Car>()
    .AsQueryable()
    .Where(x => x.Model.Contains(term));
```

While you could write one custom function for `IQueryable<Person>` and another for `IQueryable<Car>`, the following function adds this filtering to any existing query, irrespective of the specific element type.

Example

```

// using static System.Linq.Expressions.Expression;

IEnumerable<T> TextFilter<T>(IEnumerable<T> source, string term)
{
    if (string.IsNullOrEmpty(term)) { return source; }

    // T is a compile-time placeholder for the element type of the query.
    Type elementType = typeof(T);

    // Get all the string properties on this specific type.
    PropertyInfo[] stringProperties =
        elementType.GetProperties()
            .Where(x => x.PropertyType == typeof(string))
            .ToArray();
    if (!stringProperties.Any()) { return source; }

    // Get the right overload of String.Contains
    MethodInfo containsMethod = typeof(string).GetMethod("Contains", new[] { typeof(string) });

    // Create a parameter for the expression tree:
    // the 'x' in 'x => x.PropertyName.Contains("term")'
    // The type of this parameter is the query's element type
    ParameterExpression prm = Parameter(elementType);

    // Map each property to an expression tree node
    IEnumerable<Expression> expressions = stringProperties
        .Select(prp =>
            // For each property, we have to construct an expression tree node like
            x.PropertyName.Contains("term")
            Call(
                Property(
                    prm,
                    prp
                ),
                containsMethod,
                Constant(term) // "term"
            )
        );

    // Combine all the resultant expression nodes using ||
    Expression body = expressions
        .Aggregate(
            (prev, current) => Or(prev, current)
        );

    // Wrap the expression body in a compile-time-typed lambda expression
    Expression<Func<T, bool>> lambda = Lambda<Func<T, bool>>(body, prm);

    // Because the lambda is compile-time-typed (albeit with a generic parameter), we can use it with the
    Where method
    return source.Where(lambda);
}

```

Because the `TextFilter` function takes and returns an `IQueryable<T>` (and not just an `IQueryable`), you can add further compile-time-typed query elements after the text filter.

```

var qry = TextFilter(
    new List<Person>().AsQueryable(),
    "abcd"
)
.Where(x => x.DateOfBirth < new DateTime(2001, 1, 1));

var qry1 = TextFilter(
    new List<Car>().AsQueryable(),
    "abcd"
)
.Where(x => x.Year == 2010);

```

Add method call nodes to the [IQueryable](#)'s expression tree

If you have an [IQueryable](#) instead of an [IQueryable<T>](#), you can't directly call the generic LINQ methods. One alternative is to build the inner expression tree as above, and use reflection to invoke the appropriate LINQ method while passing in the expression tree.

You could also duplicate the LINQ method's functionality, by wrapping the entire tree in a [MethodCallExpression](#) that represents a call to the LINQ method:

```

IQueryable TextFilter_Untyped(IQueryable source, string term)
{
    if (string.IsNullOrEmpty(term)) { return source; }
    Type elementType = source.ElementType;

    // The logic for building the ParameterExpression and the LambdaExpression's body is the same as in the
    // previous example,
    // but has been refactored into the constructBody function.
    (Expression? body, ParameterExpression? prm) = constructBody(elementType, term);
    if (body is null) {return source;}

    Expression filteredTree = Call(
        typeof(Queryable),
        "Where",
        new[] { elementType},
        source.Expression,
        Lambda(body, prm!)
    );

    return source.Provider.CreateQuery(filteredTree);
}

```

In this case, you don't have a compile-time `T` generic placeholder, so you'll use the [Lambda](#) overload that doesn't require compile-time type information, and which produces a [LambdaExpression](#) instead of an [Expression<TDelegate>](#).

The Dynamic LINQ library

Constructing expression trees using factory methods is relatively complex; it is easier to compose strings. The [Dynamic LINQ library](#) exposes a set of extension methods on [IQueryable](#) corresponding to the standard LINQ methods at [Queryable](#), and which accept strings in a [special syntax](#) instead of expression trees. The library generates the appropriate expression tree from the string, and can return the resultant translated [IQueryable](#).

For instance, the previous example could be rewritten as follows:

```
// using System.Linq.Dynamic.Core

IEnumerable TextFilter_Strings(IEnumerable source, string term) {
    if (string.IsNullOrEmpty(term)) { return source; }

    var elementType = source.ElementType;

    // Get all the string property names on this specific type.
    var stringProperties =
        elementType.GetProperties()
            .Where(x => x.PropertyType == typeof(string))
            .ToArray();
    if (!stringProperties.Any()) { return source; }

    // Build the string expression
    string filterExpr = string.Join(
        " || ",
        stringProperties.Select(prp => $"{prp.Name}.Contains(@0)")
    );

    return source.Where(filterExpr, term);
}
```

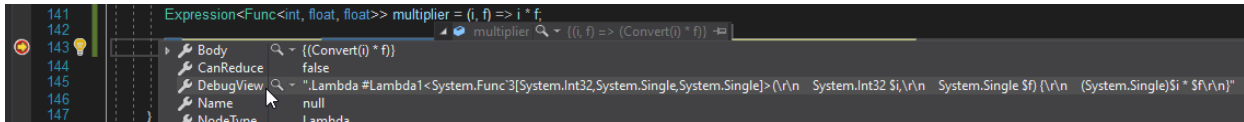
See also

- [Expression Trees \(C#\)](#)
- [How to execute expression trees \(C#\)](#)
- [Dynamically specify predicate filters at run time](#)

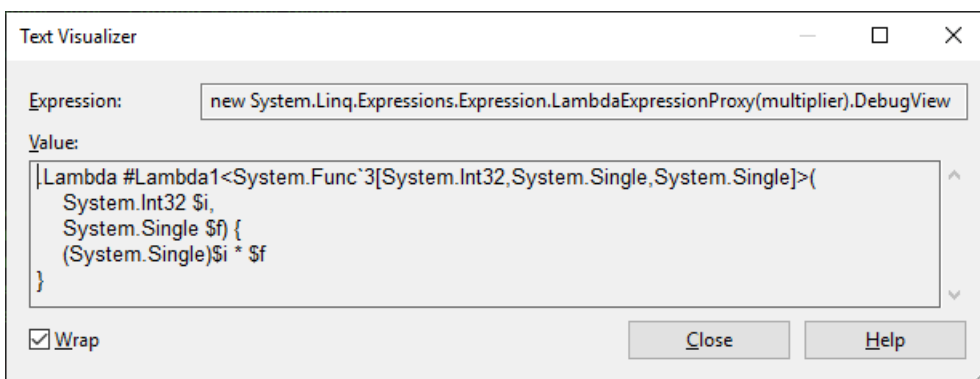
Debugging Expression Trees in Visual Studio (C#)

12/28/2021 • 2 minutes to read • [Edit Online](#)

You can analyze the structure and content of expression trees when you debug your applications. To get a quick overview of the expression tree structure, you can use the `DebugView` property, which represents expression trees [using a special syntax](#). (Note that `DebugView` is available only in debug mode.)

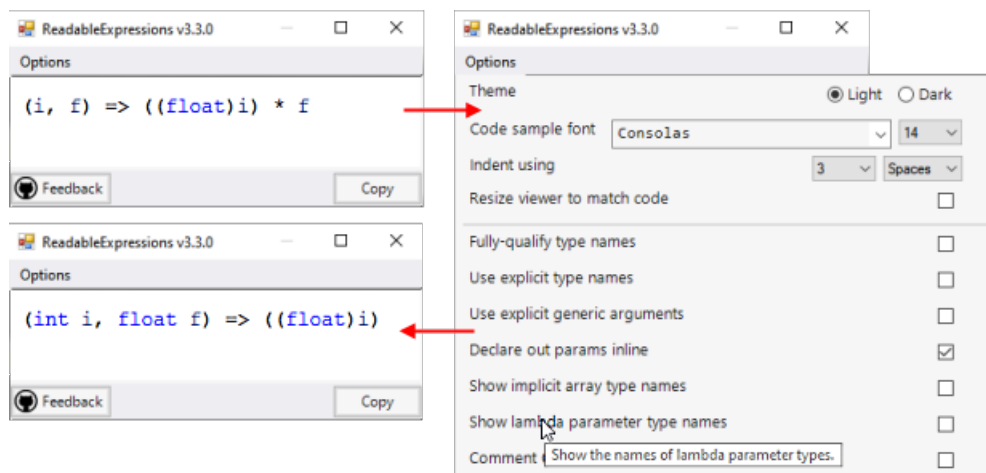


Since `DebugView` is a string, you can use the [built-in Text Visualizer](#) to view it across multiple lines, by selecting **Text Visualizer** from the magnifying glass icon next to the `DebugView` label.

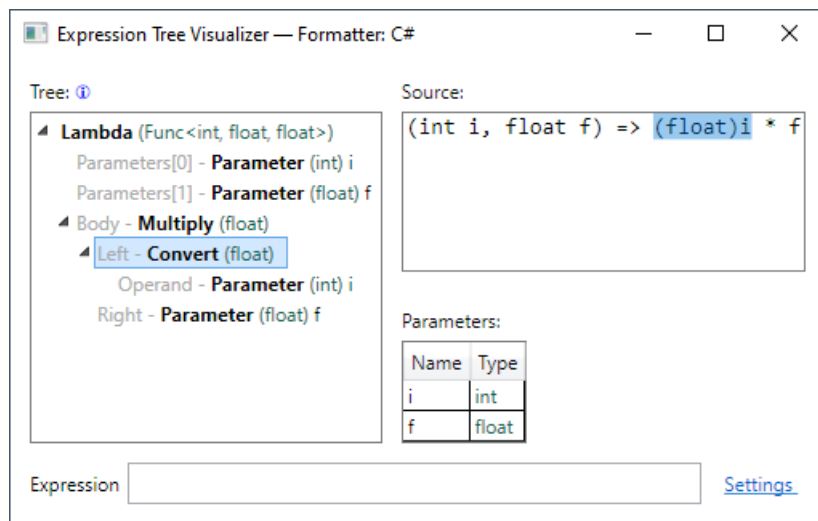


Alternatively, you can install and use [a custom visualizer](#) for expression trees, such as:

- [Readable Expressions](#) (MIT license, available at the [Visual Studio Marketplace](#)), renders the expression tree as themeable C# code, with various rendering options:



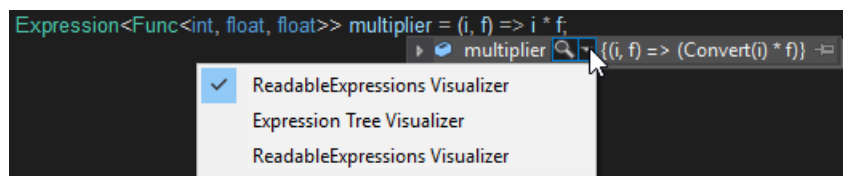
- [Expression Tree Visualizer](#) (MIT license) provides a tree view of the expression tree and its individual nodes:



To open a visualizer for an expression tree

1. Click the magnifying glass icon that appears next to the expression tree in **DataTips**, a **Watch** window, the **Autos** window, or the **Locals** window.

A list of available visualizers is displayed.:



2. Click the visualizer you want to use.

See also

- [Expression Trees \(C#\)](#)
- [Debugging in Visual Studio](#)
- [Create Custom Visualizers](#)
- [DebugView](#) syntax

DebugView syntax

12/28/2021 • 2 minutes to read • [Edit Online](#)

The **DebugView** property (available only when debugging) provides a string rendering of expression trees. Most of the syntax is fairly straightforward to understand; the special cases are described in the following sections.

Each example is followed by a block comment, containing the **DebugView**.

ParameterExpression

ParameterExpression variable names are displayed with a `$` symbol at the beginning.

If a parameter does not have a name, it is assigned an automatically generated name, such as `$var1` or `$var2`.

Examples

```
ParameterExpression numParam = Expression.Parameter(typeof(int), "num");
/*
    $num
*/

ParameterExpression numParam = Expression.Parameter(typeof(int));
/*
    $var1
*/
```

ConstantExpression

For **ConstantExpression** objects that represent integer values, strings, and `null`, the value of the constant is displayed.

For numeric types that have standard suffixes as C# literals, the suffix is added to the value. The following table shows the suffixes associated with various numeric types.

TYPE	KEYWORD	SUFFIX
System.UInt32	uint	U
System.Int64	long	L
System.UInt64	ulong	UL
System.Double	double	D
System.Single	float	F
System.Decimal	decimal	M

Examples

```
int num = 10;
ConstantExpression expr = Expression.Constant(num);
/*
    10
*/

double num = 10;
ConstantExpression expr = Expression.Constant(num);
/*
    10D
*/
```

BlockExpression

If the type of a [BlockExpression](#) object differs from the type of the last expression in the block, the type is displayed within angle brackets (< and >). Otherwise, the type of the [BlockExpression](#) object is not displayed.

Examples

```
BlockExpression block = Expression.Block(Expression.Constant("test"));
/*
    .Block() {
        "test"
    }
*/

BlockExpression block = Expression.Block(typeof(Object), Expression.Constant("test"));
/*
    .Block<System.Object>() {
        "test"
    }
*/
```

LambdaExpression

[LambdaExpression](#) objects are displayed together with their delegate types.

If a lambda expression does not have a name, it is assigned an automatically generated name, such as `#Lambda1` or `#Lambda2`.

Examples

```
LambdaExpression lambda = Expression.Lambda<Func<int>>(Expression.Constant(1));
/*
    .Lambda #Lambda1<System.Func'1[System.Int32]>() {
        1
    }
*/

LambdaExpression lambda = Expression.Lambda<Func<int>>(Expression.Constant(1), "SampleLambda", null);
/*
    .Lambda #SampleLambda<System.Func'1[System.Int32]>() {
        1
    }
*/
```

LabelExpression

If you specify a default value for the [LabelExpression](#) object, this value is displayed before the [LabelTarget](#) object.

The `.Label` token indicates the start of the label. The `.LabelTarget` token indicates the destination of the target to jump to.

If a label does not have a name, it is assigned an automatically generated name, such as `#Label1` or `#Label2`.

Examples

```
LabelTarget target = Expression.Label(typeof(int), "SampleLabel");
BlockExpression block = Expression.Block(
    Expression.Goto(target, Expression.Constant(0)),
    Expression.Label(target, Expression.Constant(-1))
);
/*
    .Block() {
        .Goto SampleLabel { 0 };
        .Label
            -1
        .LabelTarget SampleLabel:
    }
*/

LabelTarget target = Expression.Label();
BlockExpression block = Expression.Block(
    Expression.Goto(target),
    Expression.Label(target)
);
/*
    .Block() {
        .Goto #Label1 { };
        .Label
        .LabelTarget #Label1:
    }
*/
```

Checked Operators

Checked operators are displayed with the `#` symbol in front of the operator. For example, the checked addition operator is displayed as `#+`.

Examples

```
Expression expr = Expression.AddChecked( Expression.Constant(1), Expression.Constant(2));
/*
    1 #+ 2
*/

Expression expr = Expression.ConvertChecked( Expression.Constant(10.0), typeof(int));
/*
    #(System.Int32)10D
*/
```

Iterators (C#)

12/28/2021 • 7 minutes to read • [Edit Online](#)

An *iterator* can be used to step through collections such as lists and arrays.

An iterator method or `get` accessor performs a custom iteration over a collection. An iterator method uses the `yield return` statement to return each element one at a time. When a `yield return` statement is reached, the current location in code is remembered. Execution is restarted from that location the next time the iterator function is called.

You consume an iterator from client code by using a `foreach` statement or by using a LINQ query.

In the following example, the first iteration of the `foreach` loop causes execution to proceed in the `SomeNumbers` iterator method until the first `yield return` statement is reached. This iteration returns a value of 3, and the current location in the iterator method is retained. On the next iteration of the loop, execution in the iterator method continues from where it left off, again stopping when it reaches a `yield return` statement. This iteration returns a value of 5, and the current location in the iterator method is again retained. The loop completes when the end of the iterator method is reached.

```
static void Main()
{
    foreach (int number in SomeNumbers())
    {
        Console.Write(number.ToString() + " ");
    }
    // Output: 3 5 8
    Console.ReadKey();
}

public static System.Collections.IEnumerable SomeNumbers()
{
    yield return 3;
    yield return 5;
    yield return 8;
}
```

The return type of an iterator method or `get` accessor can be `IEnumerable`, `IEnumerable<T>`, `IEnumerator`, or `IEnumerator<T>`.

You can use a `yield break` statement to end the iteration.

NOTE

For all examples in this topic except the Simple Iterator example, include `using` directives for the `System.Collections` and `System.Collections.Generic` namespaces.

Simple Iterator

The following example has a single `yield return` statement that is inside a `for` loop. In `Main`, each iteration of the `foreach` statement body creates a call to the iterator function, which proceeds to the next `yield return` statement.

```

static void Main()
{
    foreach (int number in EvenSequence(5, 18))
    {
        Console.Write(number.ToString() + " ");
    }
    // Output: 6 8 10 12 14 16 18
    Console.ReadKey();
}

public static System.Collections.Generic.IEnumerable<int>
EvenSequence(int firstNumber, int lastNumber)
{
    // Yield even numbers in the range.
    for (int number = firstNumber; number <= lastNumber; number++)
    {
        if (number % 2 == 0)
        {
            yield return number;
        }
    }
}

```

Creating a Collection Class

In the following example, the `DaysOfTheWeek` class implements the `IEnumerable` interface, which requires a `GetEnumerator` method. The compiler implicitly calls the `GetEnumerator` method, which returns an `IEnumerator`.

The `GetEnumerator` method returns each string one at a time by using the `yield return` statement.

```

static void Main()
{
    DaysOfTheWeek days = new DaysOfTheWeek();

    foreach (string day in days)
    {
        Console.Write(day + " ");
    }
    // Output: Sun Mon Tue Wed Thu Fri Sat
    Console.ReadKey();
}

public class DaysOfTheWeek : IEnumerable
{
    private string[] days = { "Sun", "Mon", "Tue", "Wed", "Thu", "Fri", "Sat" };

    public IEnumerator GetEnumerator()
    {
        for (int index = 0; index < days.Length; index++)
        {
            // Yield each day of the week.
            yield return days[index];
        }
    }
}

```

The following example creates a `Zoo` class that contains a collection of animals.

The `foreach` statement that refers to the class instance (`theZoo`) implicitly calls the `GetEnumerator` method. The `foreach` statements that refer to the `Birds` and `Mammals` properties use the `AnimalsForType` named iterator method.

```

static void Main()
{
    Zoo theZoo = new Zoo();

    theZoo.AddMammal("Whale");
    theZoo.AddMammal("Rhinceros");
    theZoo.AddBird("Penguin");
    theZoo.AddBird("Warbler");

    foreach (string name in theZoo)
    {
        Console.Write(name + " ");
    }
    Console.WriteLine();
    // Output: Whale Rhinceros Penguin Warbler

    foreach (string name in theZoo.Birds)
    {
        Console.Write(name + " ");
    }
    Console.WriteLine();
    // Output: Penguin Warbler

    foreach (string name in theZoo.Mammals)
    {
        Console.Write(name + " ");
    }
    Console.WriteLine();
    // Output: Whale Rhinceros

    Console.ReadKey();
}

public class Zoo : IEnumerable
{
    // Private members.
    private List<Animal> animals = new List<Animal>();

    // Public methods.
    public void AddMammal(string name)
    {
        animals.Add(new Animal { Name = name, Type = Animal.TypeEnum.Mammal });
    }

    public void AddBird(string name)
    {
        animals.Add(new Animal { Name = name, Type = Animal.TypeEnum.Bird });
    }

    public IEnumerator GetEnumerator()
    {
        foreach (Animal theAnimal in animals)
        {
            yield return theAnimal.Name;
        }
    }

    // Public members.
    public IEnumerable Mammals
    {
        get { return AnimalsForType(Animal.TypeEnum.Mammal); }
    }

    public IEnumerable Birds
    {
        get { return AnimalsForType(Animal.TypeEnum.Bird); }
    }

    // Private methods.

```

```
// Private method.
private IEnumerable AnimalsForType(Animal.TypeEnum type)
{
    foreach (Animal theAnimal in animals)
    {
        if (theAnimal.Type == type)
        {
            yield return theAnimal.Name;
        }
    }
}

// Private class.
private class Animal
{
    public enum TypeEnum { Bird, Mammal }

    public string Name { get; set; }
    public TypeEnum Type { get; set; }
}
}
```

Using Iterators with a Generic List

In the following example, the `Stack<T>` generic class implements the `IEnumerable<T>` generic interface. The `Push` method assigns values to an array of type `T`. The `GetEnumerator` method returns the array values by using the `yield return` statement.

In addition to the generic `GetEnumerator` method, the non-generic `GetEnumerator` method must also be implemented. This is because `IEnumerable<T>` inherits from `IEnumerable`. The non-generic implementation defers to the generic implementation.

The example uses named iterators to support various ways of iterating through the same collection of data. These named iterators are the `TopToBottom` and `BottomToTop` properties, and the `TopN` method.

The `BottomToTop` property uses an iterator in a `get` accessor.

```
static void Main()
{
    Stack<int> theStack = new Stack<int>();

    // Add items to the stack.
    for (int number = 0; number <= 9; number++)
    {
        theStack.Push(number);
    }

    // Retrieve items from the stack.
    // foreach is allowed because theStack implements IEnumerable<int>.
    foreach (int number in theStack)
    {
        Console.Write("{0} ", number);
    }
    Console.WriteLine();
    // Output: 9 8 7 6 5 4 3 2 1 0

    // foreach is allowed, because theStack.TopToBottom returns IEnumerable(Of Integer).
    foreach (int number in theStack.TopToBottom)
    {
        Console.Write("{0} ", number);
    }
    Console.WriteLine();
    // Output: 9 8 7 6 5 4 3 2 1 0

    foreach (int number in theStack.BottomToTop)
```



```

        foreach (int number in theStack.Reverse())
        {
            Console.Write("{0} ", number);
        }
        Console.WriteLine();
        // Output: 0 1 2 3 4 5 6 7 8 9

        foreach (int number in theStack.TopN(7))
        {
            Console.Write("{0} ", number);
        }
        Console.WriteLine();
        // Output: 9 8 7 6 5 4 3

        Console.ReadKey();
    }
}

public class Stack<T> : IEnumerable<T>
{
    private T[] values = new T[100];
    private int top = 0;

    public void Push(T t)
    {
        values[top] = t;
        top++;
    }
    public T Pop()
    {
        top--;
        return values[top];
    }

    // This method implements the GetEnumerator method. It allows
    // an instance of the class to be used in a foreach statement.
    public IEnumerator<T> GetEnumerator()
    {
        for (int index = top - 1; index >= 0; index--)
        {
            yield return values[index];
        }
    }

    IEnumerator IEnumerable.GetEnumerator()
    {
        return GetEnumerator();
    }

    public IEnumerable<T> TopToBottom
    {
        get { return this; }
    }

    public IEnumerable<T> BottomToTop
    {
        get
        {
            for (int index = 0; index <= top - 1; index++)
            {
                yield return values[index];
            }
        }
    }

    public IEnumerable<T> TopN(int itemsFromTop)
    {
        // Return less than itemsFromTop if necessary.
        int startIndex = itemsFromTop >= top ? 0 : top - itemsFromTop;
        for (int index = top - 1; index >= startIndex; index--)

```

```

        for (int index = top - 1; index >= start index; index--)
        {
            yield return values[index];
        }
    }
}

```

Syntax Information

An iterator can occur as a method or `get` accessor. An iterator cannot occur in an event, instance constructor, static constructor, or static finalizer.

An implicit conversion must exist from the expression type in the `yield return` statement to the type argument for the `IEnumerable<T>` returned by the iterator.

In C#, an iterator method cannot have any `in`, `ref`, or `out` parameters.

In C#, `yield` is not a reserved word and has special meaning only when it is used before a `return` or `break` keyword.

Technical Implementation

Although you write an iterator as a method, the compiler translates it into a nested class that is, in effect, a state machine. This class keeps track of the position of the iterator as long the `foreach` loop in the client code continues.

To see what the compiler does, you can use the `Ildasm.exe` tool to view the Microsoft intermediate language code that's generated for an iterator method.

When you create an iterator for a [class](#) or [struct](#), you don't have to implement the whole [IEnumerator](#) interface. When the compiler detects the iterator, it automatically generates the `Current`, `MoveNext`, and `Dispose` methods of the [IEnumerator](#) or [IEnumerator<T>](#) interface.

On each successive iteration of the `foreach` loop (or the direct call to `IEnumerator.MoveNext`), the next iterator code body resumes after the previous `yield return` statement. It then continues to the next `yield return` statement until the end of the iterator body is reached, or until a `yield break` statement is encountered.

Iterators don't support the [IEnumerator.Reset](#) method. To reiterate from the start, you must obtain a new iterator. Calling [Reset](#) on the iterator returned by an iterator method throws a [NotSupportedException](#).

For additional information, see the [C# Language Specification](#).

Use of Iterators

Iterators enable you to maintain the simplicity of a `foreach` loop when you need to use complex code to populate a list sequence. This can be useful when you want to do the following:

- Modify the list sequence after the first `foreach` loop iteration.
- Avoid fully loading a large list before the first iteration of a `foreach` loop. An example is a paged fetch to load a batch of table rows. Another example is the [EnumerateFiles](#) method, which implements iterators in .NET.
- Encapsulate building the list in the iterator. In the iterator method, you can build the list and then yield each result in a loop.

See also

- `System.Collections.Generic`
- `IEnumerable<T>`
- `foreach`, `in`
- `yield`
- Using `foreach` with Arrays
- Generics

Language Integrated Query (LINQ) (C#)

12/28/2021 • 3 minutes to read • [Edit Online](#)

Language-Integrated Query (LINQ) is the name for a set of technologies based on the integration of query capabilities directly into the C# language. Traditionally, queries against data are expressed as simple strings without type checking at compile time or IntelliSense support. Furthermore, you have to learn a different query language for each type of data source: SQL databases, XML documents, various Web services, and so on. With LINQ, a query is a first-class language construct, just like classes, methods, events. You write queries against strongly typed collections of objects by using language keywords and familiar operators. The LINQ family of technologies provides a consistent query experience for objects (LINQ to Objects), relational databases (LINQ to SQL), and XML (LINQ to XML).

For a developer who writes queries, the most visible "language-integrated" part of LINQ is the query expression. Query expressions are written in a declarative *query syntax*. By using query syntax, you can perform filtering, ordering, and grouping operations on data sources with a minimum of code. You use the same basic query expression patterns to query and transform data in SQL databases, ADO.NET Datasets, XML documents and streams, and .NET collections.

You can write LINQ queries in C# for SQL Server databases, XML documents, ADO.NET Datasets, and any collection of objects that supports [IEnumerable](#) or the generic [IEnumerable<T>](#) interface. LINQ support is also provided by third parties for many Web services and other database implementations.

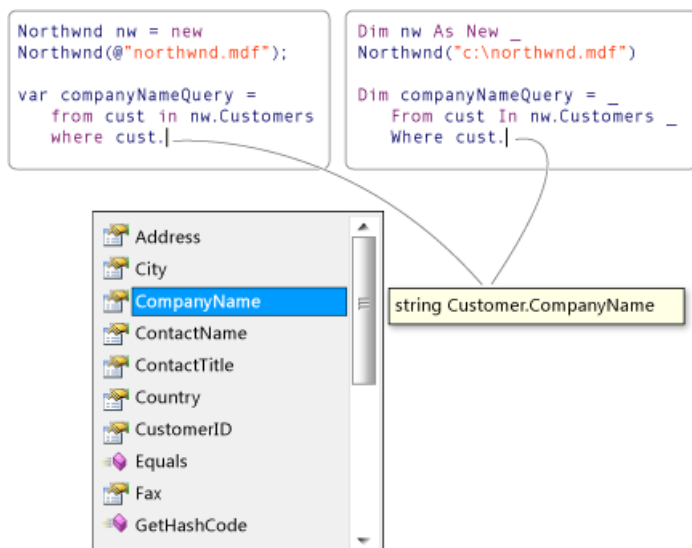
The following example shows the complete query operation. The complete operation includes creating a data source, defining the query expression, and executing the query in a `foreach` statement.

```
class LINQQueryExpressions
{
    static void Main()
    {
        // Specify the data source.
        int[] scores = new int[] { 97, 92, 81, 60 };

        // Define the query expression.
        IEnumerable<int> scoreQuery =
            from score in scores
            where score > 80
            select score;

        // Execute the query.
        foreach (int i in scoreQuery)
        {
            Console.Write(i + " ");
        }
    }
}
// Output: 97 92 81
```

The following illustration from Visual Studio shows a partially-completed LINQ query against a SQL Server database in both C# and Visual Basic with full type checking and IntelliSense support:



Query expression overview

- Query expressions can be used to query and to transform data from any LINQ-enabled data source. For example, a single query can retrieve data from a SQL database, and produce an XML stream as output.
- Query expressions are easy to grasp because they use many familiar C# language constructs.
- The variables in a query expression are all strongly typed, although in many cases you do not have to provide the type explicitly because the compiler can infer it. For more information, see [Type relationships in LINQ query operations](#).
- A query is not executed until you iterate over the query variable, for example, in a `foreach` statement. For more information, see [Introduction to LINQ queries](#).
- At compile time, query expressions are converted to Standard Query Operator method calls according to the rules set forth in the C# specification. Any query that can be expressed by using query syntax can also be expressed by using method syntax. However, in most cases query syntax is more readable and concise. For more information, see [C# language specification](#) and [Standard query operators overview](#).
- As a rule when you write LINQ queries, we recommend that you use query syntax whenever possible and method syntax whenever necessary. There is no semantic or performance difference between the two different forms. Query expressions are often more readable than equivalent expressions written in method syntax.
- Some query operations, such as [Count](#) or [Max](#), have no equivalent query expression clause and must therefore be expressed as a method call. Method syntax can be combined with query syntax in various ways. For more information, see [Query syntax and method syntax in LINQ](#).
- Query expressions can be compiled to expression trees or to delegates, depending on the type that the query is applied to. [IEnumerable<T>](#) queries are compiled to delegates. [IQueryable](#) and [IQueryable<T>](#) queries are compiled to expression trees. For more information, see [Expression trees](#).

Next steps

To learn more details about LINQ, start by becoming familiar with some basic concepts in [Query expression basics](#), and then read the documentation for the LINQ technology in which you are interested:

- XML documents: [LINQ to XML](#)
- ADO.NET Entity Framework: [LINQ to entities](#)
- .NET collections, files, strings and so on: [LINQ to objects](#)

To gain a deeper understanding of LINQ in general, see [LINQ in C#](#).

To start working with LINQ in C#, see the tutorial [Working with LINQ](#).

Introduction to LINQ Queries (C#)

12/28/2021 • 6 minutes to read • [Edit Online](#)

A *query* is an expression that retrieves data from a data source. Queries are usually expressed in a specialized query language. Different languages have been developed over time for the various types of data sources, for example SQL for relational databases and XQuery for XML. Therefore, developers have had to learn a new query language for each type of data source or data format that they must support. LINQ simplifies this situation by offering a consistent model for working with data across various kinds of data sources and formats. In a LINQ query, you are always working with objects. You use the same basic coding patterns to query and transform data in XML documents, SQL databases, ADO.NET Datasets, .NET collections, and any other format for which a LINQ provider is available.

Three Parts of a Query Operation

All LINQ query operations consist of three distinct actions:

1. Obtain the data source.
2. Create the query.
3. Execute the query.

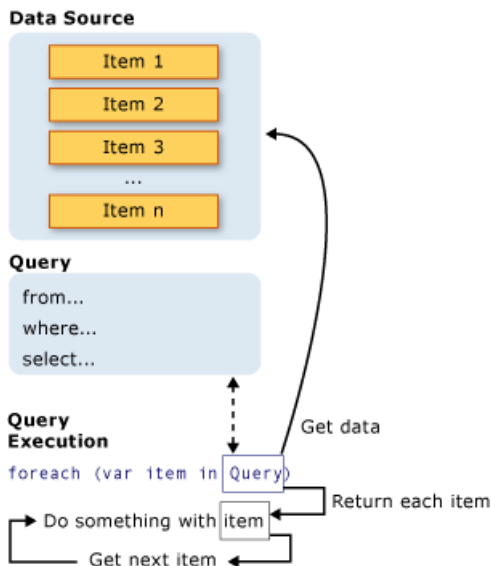
The following example shows how the three parts of a query operation are expressed in source code. The example uses an integer array as a data source for convenience; however, the same concepts apply to other data sources also. This example is referred to throughout the rest of this topic.

```
class IntroToLINQ
{
    static void Main()
    {
        // The Three Parts of a LINQ Query:
        // 1. Data source.
        int[] numbers = new int[7] { 0, 1, 2, 3, 4, 5, 6 };

        // 2. Query creation.
        // numQuery is an IEnumerable<int>
        var numQuery =
            from num in numbers
            where (num % 2) == 0
            select num;

        // 3. Query execution.
        foreach (int num in numQuery)
        {
            Console.WriteLine("{0,1} ", num);
        }
    }
}
```

The following illustration shows the complete query operation. In LINQ, the execution of the query is distinct from the query itself. In other words, you have not retrieved any data just by creating a query variable.



The Data Source

In the previous example, because the data source is an array, it implicitly supports the generic `IEnumerable<T>` interface. This fact means it can be queried with LINQ. A query is executed in a `foreach` statement, and `foreach` requires `IEnumerable` or `IEnumerable<T>`. Types that support `IEnumerable<T>` or a derived interface such as the generic `IQueryable<T>` are called *queryable types*.

A queryable type requires no modification or special treatment to serve as a LINQ data source. If the source data is not already in memory as a queryable type, the LINQ provider must represent it as such. For example, LINQ to XML loads an XML document into a queryable `XElement` type:

```
// Create a data source from an XML document.
// using System.Xml.Linq;
XElement contacts = XElement.Load(@"c:\myContactList.xml");
```

With LINQ to SQL, you first create an object-relational mapping at design time either manually or by using the [LINQ to SQL Tools in Visual Studio](#). You write your queries against the objects, and at run-time LINQ to SQL handles the communication with the database. In the following example, `Customers` represents a specific table in the database, and the type of the query result, `IQueryable<T>`, derives from `IEnumerable<T>`.

```
Northwnd db = new Northwnd(@"c:\northwnd.mdf");

// Query for customers in London.
IQueryable<Customer> custQuery =
    from cust in db.Customers
    where cust.City == "London"
    select cust;
```

For more information about how to create specific types of data sources, see the documentation for the various LINQ providers. However, the basic rule is very simple: a LINQ data source is any object that supports the generic `IEnumerable<T>` interface, or an interface that inherits from it.

NOTE

Types such as `ArrayList` that support the non-generic `IEnumerable` interface can also be used as a LINQ data source. For more information, see [How to query an ArrayList with LINQ \(C#\)](#).

The Query

The query specifies what information to retrieve from the data source or sources. Optionally, a query also specifies how that information should be sorted, grouped, and shaped before it is returned. A query is stored in a query variable and initialized with a query expression. To make it easier to write queries, C# has introduced new query syntax.

The query in the previous example returns all the even numbers from the integer array. The query expression contains three clauses: `from`, `where` and `select`. (If you are familiar with SQL, you will have noticed that the ordering of the clauses is reversed from the order in SQL.) The `from` clause specifies the data source, the `where` clause applies the filter, and the `select` clause specifies the type of the returned elements. These and the other query clauses are discussed in detail in the [Language Integrated Query \(LINQ\)](#) section. For now, the important point is that in LINQ, the query variable itself takes no action and returns no data. It just stores the information that is required to produce the results when the query is executed at some later point. For more information about how queries are constructed behind the scenes, see [Standard Query Operators Overview \(C#\)](#).

NOTE

Queries can also be expressed by using method syntax. For more information, see [Query Syntax and Method Syntax in LINQ](#).

Query Execution

Deferred Execution

As stated previously, the query variable itself only stores the query commands. The actual execution of the query is deferred until you iterate over the query variable in a `foreach` statement. This concept is referred to as *deferred execution* and is demonstrated in the following example:

```
// Query execution.
foreach (int num in numQuery)
{
    Console.WriteLine("{0,1} ", num);
}
```

The `foreach` statement is also where the query results are retrieved. For example, in the previous query, the iteration variable `num` holds each value (one at a time) in the returned sequence.

Because the query variable itself never holds the query results, you can execute it as often as you like. For example, you may have a database that is being updated continually by a separate application. In your application, you could create one query that retrieves the latest data, and you could execute it repeatedly at some interval to retrieve different results every time.

Forcing Immediate Execution

Queries that perform aggregation functions over a range of source elements must first iterate over those elements. Examples of such queries are `Count`, `Max`, `Average`, and `First`. These execute without an explicit `foreach` statement because the query itself must use `foreach` in order to return a result. Note also that these types of queries return a single value, not an `IEnumerable` collection. The following query returns a count of the even numbers in the source array:


```
var evenNumQuery =  
    from num in numbers  
    where (num % 2) == 0  
    select num;  
  
int evenNumCount = evenNumQuery.Count();
```

To force immediate execution of any query and cache its results, you can call the [ToList](#) or [ToArray](#) methods.

```
List<int> numQuery2 =  
    (from num in numbers  
     where (num % 2) == 0  
     select num).ToList();  
  
// or like this:  
// numQuery3 is still an int[]  
  
var numQuery3 =  
    (from num in numbers  
     where (num % 2) == 0  
     select num).ToArray();
```

You can also force execution by putting the `foreach` loop immediately after the query expression. However, by calling `ToList` or `ToArray` you also cache all the data in a single collection object.

See also

- [Getting Started with LINQ in C#](#)
- [Walkthrough: Writing Queries in C#](#)
- [Language Integrated Query \(LINQ\)](#)
- [foreach, in](#)
- [Query Keywords \(LINQ\)](#)

LINQ and Generic Types (C#)

12/28/2021 • 2 minutes to read • [Edit Online](#)

LINQ queries are based on generic types, which were introduced in version 2.0 of .NET Framework. You do not need an in-depth knowledge of generics before you can start writing queries. However, you may want to understand two basic concepts:

1. When you create an instance of a generic collection class such as `List<T>`, you replace the "T" with the type of objects that the list will hold. For example, a list of strings is expressed as `List<string>`, and a list of `Customer` objects is expressed as `List<Customer>`. A generic list is strongly typed and provides many benefits over collections that store their elements as `Object`. If you try to add a `Customer` to a `List<string>`, you will get an error at compile time. It is easy to use generic collections because you do not have to perform run-time type-casting.
2. `IEnumerable<T>` is the interface that enables generic collection classes to be enumerated by using the `foreach` statement. Generic collection classes support `IEnumerable<T>` just as non-generic collection classes such as `ArrayList` support `IEnumerable`.

For more information about generics, see [Generics](#).

`IEnumerable<T>` variables in LINQ Queries

LINQ query variables are typed as `IEnumerable<T>` or a derived type such as `IQueryable<T>`. When you see a query variable that is typed as `IEnumerable<Customer>`, it just means that the query, when it is executed, will produce a sequence of zero or more `Customer` objects.

```
IEnumerable<Customer> customerQuery =
    from cust in customers
    where cust.City == "London"
    select cust;

foreach (Customer customer in customerQuery)
{
    Console.WriteLine(customer.LastName + ", " + customer.FirstName);
}
```

For more information, see [Type Relationships in LINQ Query Operations](#).

Letting the Compiler Handle Generic Type Declarations

If you prefer, you can avoid generic syntax by using the `var` keyword. The `var` keyword instructs the compiler to infer the type of a query variable by looking at the data source specified in the `from` clause. The following example produces the same compiled code as the previous example:

```
var customerQuery2 =  
    from cust in customers  
    where cust.City == "London"  
    select cust;  
  
foreach(var customer in customerQuery2)  
{  
    Console.WriteLine(customer.LastName + ", " + customer.FirstName);  
}
```

The `var` keyword is useful when the type of the variable is obvious or when it is not that important to explicitly specify nested generic types such as those that are produced by group queries. In general, we recommend that if you use `var`, realize that it can make your code more difficult for others to read. For more information, see [Implicitly Typed Local Variables](#).

See also

- [Generics](#)

Basic LINQ Query Operations (C#)

12/28/2021 • 5 minutes to read • [Edit Online](#)

This topic gives a brief introduction to LINQ query expressions and some of the typical kinds of operations that you perform in a query. More detailed information is in the following topics:

[LINQ Query Expressions](#)

[Standard Query Operators Overview \(C#\)](#)

[Walkthrough: Writing Queries in C#](#)

NOTE

If you already are familiar with a query language such as SQL or XQuery, you can skip most of this topic. Read about the "`from` clause" in the next section to learn about the order of clauses in LINQ query expressions.

Obtaining a Data Source

In a LINQ query, the first step is to specify the data source. In C# as in most programming languages a variable must be declared before it can be used. In a LINQ query, the `from` clause comes first in order to introduce the data source (`customers`) and the *range variable* (`cust`).

```
//queryAllCustomers is an IEnumerable<Customer>
var queryAllCustomers = from cust in customers
                        select cust;
```

The range variable is like the iteration variable in a `foreach` loop except that no actual iteration occurs in a query expression. When the query is executed, the range variable will serve as a reference to each successive element in `customers`. Because the compiler can infer the type of `cust`, you do not have to specify it explicitly. Additional range variables can be introduced by a `let` clause. For more information, see [let clause](#).

NOTE

For non-generic data sources such as [ArrayList](#), the range variable must be explicitly typed. For more information, see [How to query an ArrayList with LINQ \(C#\)](#) and [from clause](#).

Filtering

Probably the most common query operation is to apply a filter in the form of a Boolean expression. The filter causes the query to return only those elements for which the expression is true. The result is produced by using the `where` clause. The filter in effect specifies which elements to exclude from the source sequence. In the following example, only those `customers` who have an address in London are returned.

```
var queryLondonCustomers = from cust in customers
                           where cust.City == "London"
                           select cust;
```

You can use the familiar C# logical `AND` and `OR` operators to apply as many filter expressions as necessary in

the `where` clause. For example, to return only customers from "London" `AND` whose name is "Devon" you would write the following code:

```
where cust.City == "London" && cust.Name == "Devon"
```

To return customers from London or Paris, you would write the following code:

```
where cust.City == "London" || cust.City == "Paris"
```

For more information, see [where clause](#).

Ordering

Often it is convenient to sort the returned data. The `orderby` clause will cause the elements in the returned sequence to be sorted according to the default comparer for the type being sorted. For example, the following query can be extended to sort the results based on the `Name` property. Because `Name` is a string, the default comparer performs an alphabetical sort from A to Z.

```
var queryLondonCustomers3 =  
    from cust in customers  
    where cust.City == "London"  
    orderby cust.Name ascending  
    select cust;
```

To order the results in reverse order, from Z to A, use the `orderby...descending` clause.

For more information, see [orderby clause](#).

Grouping

The `group` clause enables you to group your results based on a key that you specify. For example you could specify that the results should be grouped by the `City` so that all customers from London or Paris are in individual groups. In this case, `cust.City` is the key.

```
// queryCustomersByCity is an IEnumerable<IGrouping<string, Customer>>  
var queryCustomersByCity =  
    from cust in customers  
    group cust by cust.City;  
  
// customerGroup is an IGrouping<string, Customer>  
foreach (var customerGroup in queryCustomersByCity)  
{  
    Console.WriteLine(customerGroup.Key);  
    foreach (Customer customer in customerGroup)  
    {  
        Console.WriteLine("    {0}", customer.Name);  
    }  
}
```

When you end a query with a `group` clause, your results take the form of a list of lists. Each element in the list is an object that has a `key` member and a list of elements that are grouped under that key. When you iterate over a query that produces a sequence of groups, you must use a nested `foreach` loop. The outer loop iterates over each group, and the inner loop iterates over each group's members.

If you must refer to the results of a group operation, you can use the `into` keyword to create an identifier that

can be queried further. The following query returns only those groups that contain more than two customers:

```
// custQuery is an IEnumerable<IGrouping<string, Customer>>
var custQuery =
    from cust in customers
    group cust by cust.City into custGroup
    where custGroup.Count() > 2
    orderby custGroup.Key
    select custGroup;
```

For more information, see [group clause](#).

Joining

Join operations create associations between sequences that are not explicitly modeled in the data sources. For example you can perform a join to find all the customers and distributors who have the same location. In LINQ the `join` clause always works against object collections instead of database tables directly.

```
var innerJoinQuery =
    from cust in customers
    join dist in distributors on cust.City equals dist.City
    select new { CustomerName = cust.Name, DistributorName = dist.Name };
```

In LINQ, you do not have to use `join` as often as you do in SQL, because foreign keys in LINQ are represented in the object model as properties that hold a collection of items. For example, a `Customer` object contains a collection of `Order` objects. Rather than performing a join, you access the orders by using dot notation:

```
from order in Customer.Orders...
```

For more information, see [join clause](#).

Selecting (Projections)

The `select` clause produces the results of the query and specifies the "shape" or type of each returned element. For example, you can specify whether your results will consist of complete `Customer` objects, just one member, a subset of members, or some completely different result type based on a computation or new object creation. When the `select` clause produces something other than a copy of the source element, the operation is called a *projection*. The use of projections to transform data is a powerful capability of LINQ query expressions. For more information, see [Data Transformations with LINQ \(C#\)](#) and [select clause](#).

See also

- [LINQ Query Expressions](#)
- [Walkthrough: Writing Queries in C#](#)
- [Query Keywords \(LINQ\)](#)
- [Anonymous Types](#)

Data Transformations with LINQ (C#)

12/28/2021 • 6 minutes to read • [Edit Online](#)

Language-Integrated Query (LINQ) is not only about retrieving data. It is also a powerful tool for transforming data. By using a LINQ query, you can use a source sequence as input and modify it in many ways to create a new output sequence. You can modify the sequence itself without modifying the elements themselves by sorting and grouping. But perhaps the most powerful feature of LINQ queries is the ability to create new types. This is accomplished in the [select](#) clause. For example, you can perform the following tasks:

- Merge multiple input sequences into a single output sequence that has a new type.
- Create output sequences whose elements consist of only one or several properties of each element in the source sequence.
- Create output sequences whose elements consist of the results of operations performed on the source data.
- Create output sequences in a different format. For example, you can transform data from SQL rows or text files into XML.

These are just several examples. Of course, these transformations can be combined in various ways in the same query. Furthermore, the output sequence of one query can be used as the input sequence for a new query.

Joining Multiple Inputs into One Output Sequence

You can use a LINQ query to create an output sequence that contains elements from more than one input sequence. The following example shows how to combine two in-memory data structures, but the same principles can be applied to combine data from XML or SQL or DataSet sources. Assume the following two class types:

```
class Student
{
    public string First { get; set; }
    public string Last { get; set; }
    public int ID { get; set; }
    public string Street { get; set; }
    public string City { get; set; }
    public List<int> Scores;
}

class Teacher
{
    public string First { get; set; }
    public string Last { get; set; }
    public int ID { get; set; }
    public string City { get; set; }
}
```

The following example shows the query:

```

class DataTransformations
{
    static void Main()
    {
        // Create the first data source.
        List<Student> students = new List<Student>()
        {
            new Student { First="Svetlana",
                          Last="Omelchenko",
                          ID=111,
                          Street="123 Main Street",
                          City="Seattle",
                          Scores= new List<int> { 97, 92, 81, 60 } },
            new Student { First="Claire",
                          Last="O'Donnell",
                          ID=112,
                          Street="124 Main Street",
                          City="Redmond",
                          Scores= new List<int> { 75, 84, 91, 39 } },
            new Student { First="Sven",
                          Last="Mortensen",
                          ID=113,
                          Street="125 Main Street",
                          City="Lake City",
                          Scores= new List<int> { 88, 94, 65, 91 } },
        };

        // Create the second data source.
        List<Teacher> teachers = new List<Teacher>()
        {
            new Teacher { First="Ann", Last="Beebe", ID=945, City="Seattle" },
            new Teacher { First="Alex", Last="Robinson", ID=956, City="Redmond" },
            new Teacher { First="Michiyo", Last="Sato", ID=972, City="Tacoma" }
        };

        // Create the query.
        var peopleInSeattle = (from student in students
                               where student.City == "Seattle"
                               select student.Last)
                               .Concat(from teacher in teachers
                                       where teacher.City == "Seattle"
                                       select teacher.Last);

        Console.WriteLine("The following students and teachers live in Seattle:");
        // Execute the query.
        foreach (var person in peopleInSeattle)
        {
            Console.WriteLine(person);
        }

        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
/* Output:
   The following students and teachers live in Seattle:
   Omelchenko
   Beebe
*/

```

For more information, see [join clause](#) and [select clause](#).

Selecting a Subset of each Source Element

There are two primary ways to select a subset of each element in the source sequence:

1. To select just one member of the source element, use the dot operation. In the following example, assume that a `Customer` object contains several public properties including a string named `City`. When executed, this query will produce an output sequence of strings.

```
var query = from cust in Customers
            select cust.City;
```

2. To create elements that contain more than one property from the source element, you can use an object initializer with either a named object or an anonymous type. The following example shows the use of an anonymous type to encapsulate two properties from each `Customer` element:

```
var query = from cust in Customer
            select new {Name = cust.Name, City = cust.City};
```

For more information, see [Object and Collection Initializers](#) and [Anonymous Types](#).

Transforming in-Memory Objects into XML

LINQ queries make it easy to transform data between in-memory data structures, SQL databases, ADO.NET Datasets and XML streams or documents. The following example transforms objects in an in-memory data structure into XML elements.

```
class XMLTransform
{
    static void Main()
    {
        // Create the data source by using a collection initializer.
        // The Student class was defined previously in this topic.
        List<Student> students = new List<Student>()
        {
            new Student {First="Svetlana", Last="Omelchenko", ID=111, Scores = new List<int>{97, 92, 81,
60}},
            new Student {First="Claire", Last="O'Donnell", ID=112, Scores = new List<int>{75, 84, 91, 39}},
            new Student {First="Sven", Last="Mortensen", ID=113, Scores = new List<int>{88, 94, 65, 91}},
        };

        // Create the query.
        var studentsToXML = new XElement("Root",
            from student in students
            let scores = string.Join(",", student.Scores)
            select new XElement("student",
                new XElement("First", student.First),
                new XElement("Last", student.Last),
                new XElement("Scores", scores)
            ) // end "student"
        ); // end "Root"

        // Execute the query.
        Console.WriteLine(studentsToXML);

        // Keep the console open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
```

The code produces the following XML output:

```
<Root>
  <student>
    <First>Svetlana</First>
    <Last>Omelchenko</Last>
    <Scores>97,92,81,60</Scores>
  </student>
  <student>
    <First>Claire</First>
    <Last>O'Donnell</Last>
    <Scores>75,84,91,39</Scores>
  </student>
  <student>
    <First>Sven</First>
    <Last>Mortensen</Last>
    <Scores>88,94,65,91</Scores>
  </student>
</Root>
```

For more information, see [Creating XML Trees in C# \(LINQ to XML\)](#).

Performing Operations on Source Elements

An output sequence might not contain any elements or element properties from the source sequence. The output might instead be a sequence of values that is computed by using the source elements as input arguments.

The following query will take a sequence of numbers that represent radii of circles, calculate the area for each radius, and return an output sequence containing strings formatted with the calculated area.

Each string for the output sequence will be formatted using [string interpolation](#). An interpolated string will have a `$` in front of the string's opening quotation mark, and operations can be performed within curly braces placed inside the interpolated string. Once those operations are performed, the results will be concatenated.

NOTE

Calling methods in query expressions is not supported if the query will be translated into some other domain. For example, you cannot call an ordinary C# method in LINQ to SQL because SQL Server has no context for it. However, you can map stored procedures to methods and call those. For more information, see [Stored Procedures](#).

```

class FormatQuery
{
    static void Main()
    {
        // Data source.
        double[] radii = { 1, 2, 3 };

        // LINQ query using method syntax.
        IEnumerable<string> output =
            radii.Select(r => $"Area for a circle with a radius of '{r}' = {r * r * Math.PI:F2}");

        /*
        // LINQ query using query syntax.
        IEnumerable<string> output =
            from rad in radii
            select $"Area for a circle with a radius of '{rad}' = {rad * rad * Math.PI:F2}";
        */

        foreach (string s in output)
        {
            Console.WriteLine(s);
        }

        // Keep the console open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
/* Output:
Area for a circle with a radius of '1' = 3.14
Area for a circle with a radius of '2' = 12.57
Area for a circle with a radius of '3' = 28.27
*/

```

See also

- [Language-Integrated Query \(LINQ\) \(C#\)](#)
- [LINQ to SQL](#)
- [LINQ to DataSet](#)
- [LINQ to XML \(C#\)](#)
- [LINQ Query Expressions](#)
- [select clause](#)

Type Relationships in LINQ Query Operations (C#)

12/28/2021 • 2 minutes to read • [Edit Online](#)

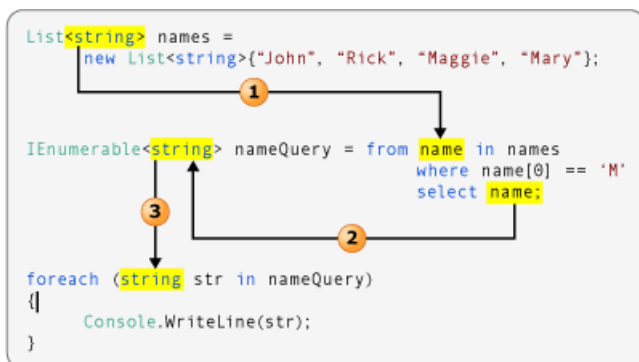
To write queries effectively, you should understand how types of the variables in a complete query operation all relate to each other. If you understand these relationships you will more easily comprehend the LINQ samples and code examples in the documentation. Furthermore, you will understand what occurs behind the scenes when variables are implicitly typed by using `var`.

LINQ query operations are strongly typed in the data source, in the query itself, and in the query execution. The type of the variables in the query must be compatible with the type of the elements in the data source and with the type of the iteration variable in the `foreach` statement. This strong typing guarantees that type errors are caught at compile time when they can be corrected before users encounter them.

In order to demonstrate these type relationships, most of the examples that follow use explicit typing for all variables. The last example shows how the same principles apply even when you use implicit typing by using `var`.

Queries that do not Transform the Source Data

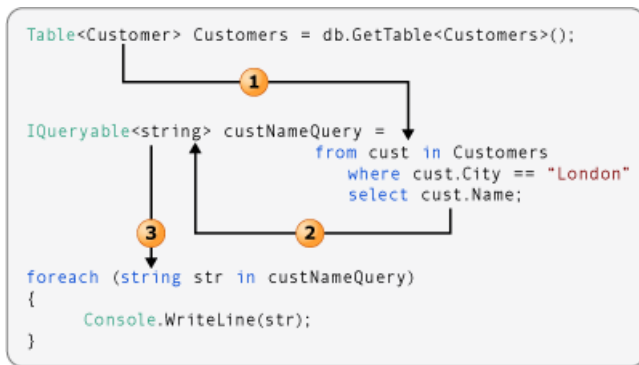
The following illustration shows a LINQ to Objects query operation that performs no transformations on the data. The source contains a sequence of strings and the query output is also a sequence of strings.



1. The type argument of the data source determines the type of the range variable.
2. The type of the object that is selected determines the type of the query variable. Here `name` is a string. Therefore, the query variable is an `IEnumerable<string>`.
3. The query variable is iterated over in the `foreach` statement. Because the query variable is a sequence of strings, the iteration variable is also a string.

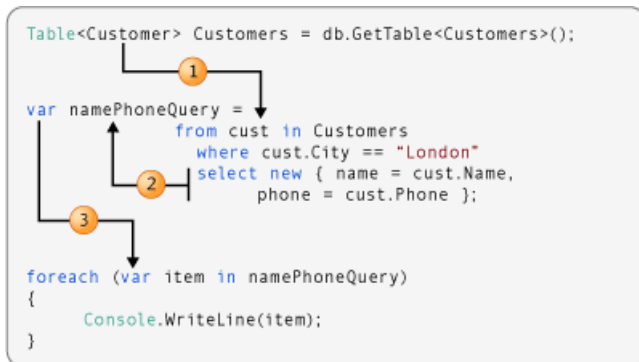
Queries that Transform the Source Data

The following illustration shows a LINQ to SQL query operation that performs a simple transformation on the data. The query takes a sequence of `Customer` objects as input, and selects only the `Name` property in the result. Because `Name` is a string, the query produces a sequence of strings as output.



1. The type argument of the data source determines the type of the range variable.
2. The `select` statement returns the `Name` property instead of the complete `Customer` object. Because `Name` is a string, the type argument of `custNameQuery` is `string`, not `Customer`.
3. Because `custNameQuery` is a sequence of strings, the `foreach` loop's iteration variable must also be a `string`.

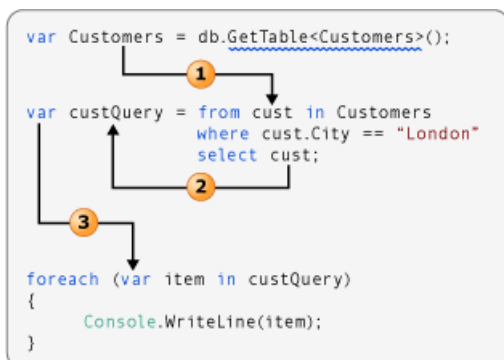
The following illustration shows a slightly more complex transformation. The `select` statement returns an anonymous type that captures just two members of the original `Customer` object.



1. The type argument of the data source is always the type of the range variable in the query.
2. Because the `select` statement produces an anonymous type, the query variable must be implicitly typed by using `var`.
3. Because the type of the query variable is implicit, the iteration variable in the `foreach` loop must also be implicit.

Letting the compiler infer type information

Although you should understand the type relationships in a query operation, you have the option to let the compiler do all the work for you. The keyword `var` can be used for any local variable in a query operation. The following illustration is similar to example number 2 that was discussed earlier. However, the compiler supplies the strong type for each variable in the query operation.



For more information about `var`, see [Implicitly Typed Local Variables](#).

Query Syntax and Method Syntax in LINQ (C#)

12/28/2021 • 4 minutes to read • [Edit Online](#)

Most queries in the introductory Language Integrated Query (LINQ) documentation are written by using the LINQ declarative query syntax. However, the query syntax must be translated into method calls for the .NET common language runtime (CLR) when the code is compiled. These method calls invoke the standard query operators, which have names such as `Where`, `Select`, `GroupBy`, `Join`, `Max`, and `Average`. You can call them directly by using method syntax instead of query syntax.

Query syntax and method syntax are semantically identical, but many people find query syntax simpler and easier to read. Some queries must be expressed as method calls. For example, you must use a method call to express a query that retrieves the number of elements that match a specified condition. You also must use a method call for a query that retrieves the element that has the maximum value in a source sequence. The reference documentation for the standard query operators in the [System.Linq](#) namespace generally uses method syntax. Therefore, even when getting started writing LINQ queries, it is useful to be familiar with how to use method syntax in queries and in query expressions themselves.

Standard Query Operator Extension Methods

The following example shows a simple *query expression* and the semantically equivalent query written as a *method-based query*.

```

class QueryVMMethodSyntax
{
    static void Main()
    {
        int[] numbers = { 5, 10, 8, 3, 6, 12};

        //Query syntax:
        IEnumerable<int> numQuery1 =
            from num in numbers
            where num % 2 == 0
            orderby num
            select num;

        //Method syntax:
        IEnumerable<int> numQuery2 = numbers.Where(num => num % 2 == 0).OrderBy(n => n);

        foreach (int i in numQuery1)
        {
            Console.Write(i + " ");
        }
        Console.WriteLine(System.Environment.NewLine);
        foreach (int i in numQuery2)
        {
            Console.Write(i + " ");
        }

        // Keep the console open in debug mode.
        Console.WriteLine(System.Environment.NewLine);
        Console.WriteLine("Press any key to exit");
        Console.ReadKey();
    }
}
/*
Output:
6 8 10 12
6 8 10 12
*/

```

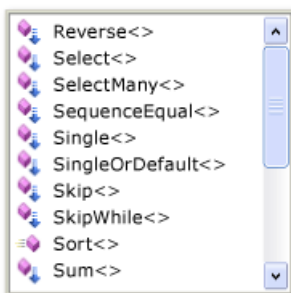
The output from the two examples is identical. You can see that the type of the query variable is the same in both forms: `IEnumerable<T>`.

To understand the method-based query, let's examine it more closely. On the right side of the expression, notice that the `where` clause is now expressed as an instance method on the `numbers` object, which as you will recall has a type of `IEnumerable<int>`. If you are familiar with the generic `IEnumerable<T>` interface, you know that it does not have a `Where` method. However, if you invoke the IntelliSense completion list in the Visual Studio IDE, you will see not only a `Where` method, but many other methods such as `Select`, `SelectMany`, `Join`, and `OrderBy`. These are all the standard query operators.

```

List<string> list = new List<string>();
list.

```



Although it looks as if `IEnumerable<T>` has been redefined to include these additional methods, in fact this is not the case. The standard query operators are implemented as a new kind of method called *extension methods*. Extensions methods "extend" an existing type; they can be called as if they were instance methods on the type.

The standard query operators extend `IEnumerable<T>` and that is why you can write `numbers.Where(...)`.

To get started using LINQ, all that you really have to know about extension methods is how to bring them into scope in your application by using the correct `using` directives. From your application's point of view, an extension method and a regular instance method are the same.

For more information about extension methods, see [Extension Methods](#). For more information about standard query operators, see [Standard Query Operators Overview \(C#\)](#). Some LINQ providers, such as LINQ to SQL and LINQ to XML, implement their own standard query operators and additional extension methods for other types besides `IEnumerable<T>`.

Lambda Expressions

In the previous example, notice that the conditional expression (`num % 2 == 0`) is passed as an in-line argument to the `Where` method: `Where(num => num % 2 == 0)`. This inline expression is called a lambda expression. It is a convenient way to write code that would otherwise have to be written in more cumbersome form as an anonymous method or a generic delegate or an expression tree. In C# `=>` is the lambda operator, which is read as "goes to". The `num` on the left of the operator is the input variable which corresponds to `num` in the query expression. The compiler can infer the type of `num` because it knows that `numbers` is a generic `IEnumerable<T>` type. The body of the lambda is just the same as the expression in query syntax or in any other C# expression or statement; it can include method calls and other complex logic. The "return value" is just the expression result.

To get started using LINQ, you do not have to use lambdas extensively. However, certain queries can only be expressed in method syntax and some of those require lambda expressions. After you become more familiar with lambdas, you will find that they are a powerful and flexible tool in your LINQ toolbox. For more information, see [Lambda Expressions](#).

Composability of Queries

In the previous code example, note that the `OrderBy` method is invoked by using the dot operator on the call to `Where`. `Where` produces a filtered sequence, and then `OrderBy` operates on that sequence by sorting it. Because queries return an `IEnumerable`, you compose them in method syntax by chaining the method calls together. This is what the compiler does behind the scenes when you write queries by using query syntax. And because a query variable does not store the results of the query, you can modify it or use it as the basis for a new query at any time, even after it has been executed.

C# Features That Support LINQ

12/28/2021 • 3 minutes to read • [Edit Online](#)

The following section introduces new language constructs introduced in C# 3.0. Although these new features are all used to a degree with LINQ queries, they are not limited to LINQ and can be used in any context where you find them useful.

Query Expressions

Query expressions use a declarative syntax similar to SQL or XQuery to query over `IEnumerable` collections. At compile time query syntax is converted to method calls to a LINQ provider's implementation of the standard query operator extension methods. Applications control the standard query operators that are in scope by specifying the appropriate namespace with a `using` directive. The following query expression takes an array of strings, groups them according to the first character in the string, and orders the groups.

```
var query = from str in stringArray
            group str by str[0] into stringGroup
            orderby stringGroup.Key
            select stringGroup;
```

For more information, see [LINQ Query Expressions](#).

Implicitly Typed Variables (var)

Instead of explicitly specifying a type when you declare and initialize a variable, you can use the `var` modifier to instruct the compiler to infer and assign the type, as shown here:

```
var number = 5;
var name = "Virginia";
var query = from str in stringArray
            where str[0] == 'm'
            select str;
```

Variables declared as `var` are just as strongly typed as variables whose type you specify explicitly. The use of `var` makes it possible to create anonymous types, but it can be used only for local variables. Arrays can also be declared with implicit typing.

For more information, see [Implicitly Typed Local Variables](#).

Object and Collection Initializers

Object and collection initializers make it possible to initialize objects without explicitly calling a constructor for the object. Initializers are typically used in query expressions when they project the source data into a new data type. Assuming a class named `Customer` with public `Name` and `Phone` properties, the object initializer can be used as in the following code:

```
var cust = new Customer { Name = "Mike", Phone = "555-1212" };
```

Continuing with our `Customer` class, assume that there is a data source called `IncomingOrders`, and that for each order with a large `OrderSize`, we would like to create a new `Customer` based off of that order. A LINQ query can

be executed on this data source and use object initialization to fill a collection:

```
var newLargeOrderCustomers = from o in IncomingOrders
    where o.OrderSize > 5
    select new Customer { Name = o.Name, Phone = o.Phone };
```

The data source may have more properties lying under the hood than the `Customer` class such as `OrderSize`, but with object initialization, the data returned from the query is molded into the desired data type; we choose the data that is relevant to our class. As a result, we now have an `IEnumerable` filled with the new `Customer`s we wanted. The above can also be written in LINQ's method syntax:

```
var newLargeOrderCustomers = IncomingOrders.Where(x => x.OrderSize > 5).Select(y => new Customer { Name = y.Name, Phone = y.Phone });
```

For more information, see:

- [Object and Collection Initializers](#)
- [Query Expression Syntax for Standard Query Operators](#)

Anonymous Types

An anonymous type is constructed by the compiler and the type name is only available to the compiler. Anonymous types provide a convenient way to group a set of properties temporarily in a query result without having to define a separate named type. Anonymous types are initialized with a new expression and an object initializer, as shown here:

```
select new {name = cust.Name, phone = cust.Phone};
```

For more information, see [Anonymous Types](#).

Extension Methods

An extension method is a static method that can be associated with a type, so that it can be called as if it were an instance method on the type. This feature enables you to, in effect, "add" new methods to existing types without actually modifying them. The standard query operators are a set of extension methods that provide LINQ query functionality for any type that implements `IEnumerable<T>`.

For more information, see [Extension Methods](#).

Lambda Expressions

A lambda expression is an inline function that uses the `=>` operator to separate input parameters from the function body and can be converted at compile time to a delegate or an expression tree. In LINQ programming, you encounter lambda expressions when you make direct method calls to the standard query operators.

For more information, see:

- [Lambda Expressions](#)
- [Expression Trees \(C#\)](#)

See also

- [Language-Integrated Query \(LINQ\) \(C#\)](#)

Walkthrough: Writing Queries in C# (LINQ)

12/28/2021 • 10 minutes to read • [Edit Online](#)

This walkthrough demonstrates the C# language features that are used to write LINQ query expressions.

Create a C# Project

NOTE

The following instructions are for Visual Studio. If you are using a different development environment, create a console project with a reference to System.Core.dll and a `using` directive for the [System.Linq](#) namespace.

To create a project in Visual Studio

1. Start Visual Studio.
2. On the menu bar, choose **File, New, Project**.

The **New Project** dialog box opens.
3. Expand **Installed**, expand **Templates**, expand **Visual C#**, and then choose **Console Application**.
4. In the **Name** text box, enter a different name or accept the default name, and then choose the **OK** button.

The new project appears in **Solution Explorer**.

5. Notice that your project has a reference to System.Core.dll and a `using` directive for the [System.Linq](#) namespace.

Create an in-Memory Data Source

The data source for the queries is a simple list of `Student` objects. Each `Student` record has a first name, last name, and an array of integers that represents their test scores in the class. Copy this code into your project. Note the following characteristics:

- The `Student` class consists of auto-implemented properties.
- Each student in the list is initialized with an object initializer.
- The list itself is initialized with a collection initializer.

This whole data structure will be initialized and instantiated without explicit calls to any constructor or explicit member access. For more information about these new features, see [Auto-Implemented Properties](#) and [Object and Collection Initializers](#).

To add the data source

- Add the `Student` class and the initialized list of students to the `Program` class in your project.

```

public class Student
{
    public string First { get; set; }
    public string Last { get; set; }
    public int ID { get; set; }
    public List<int> Scores;
}

// Create a data source by using a collection initializer.
static List<Student> students = new List<Student>
{
    new Student {First="Svetlana", Last="Omelchenko", ID=111, Scores= new List<int> {97, 92, 81, 60}},
    new Student {First="Claire", Last="O'Donnell", ID=112, Scores= new List<int> {75, 84, 91, 39}},
    new Student {First="Sven", Last="Mortensen", ID=113, Scores= new List<int> {88, 94, 65, 91}},
    new Student {First="Cesar", Last="Garcia", ID=114, Scores= new List<int> {97, 89, 85, 82}},
    new Student {First="Debra", Last="Garcia", ID=115, Scores= new List<int> {35, 72, 91, 70}},
    new Student {First="Fadi", Last="Fakhouri", ID=116, Scores= new List<int> {99, 86, 90, 94}},
    new Student {First="Hanying", Last="Feng", ID=117, Scores= new List<int> {93, 92, 80, 87}},
    new Student {First="Hugo", Last="Garcia", ID=118, Scores= new List<int> {92, 90, 83, 78}},
    new Student {First="Lance", Last="Tucker", ID=119, Scores= new List<int> {68, 79, 88, 92}},
    new Student {First="Terry", Last="Adams", ID=120, Scores= new List<int> {99, 82, 81, 79}},
    new Student {First="Eugene", Last="Zabokritski", ID=121, Scores= new List<int> {96, 85, 91, 60}},
    new Student {First="Michael", Last="Tucker", ID=122, Scores= new List<int> {94, 92, 91, 91}}
};

```

To add a new Student to the Students list

1. Add a new `Student` to the `Students` list and use a name and test scores of your choice. Try typing all the new student information in order to better learn the syntax for the object initializer.

Create the Query

To create a simple query

- In the application's `Main` method, create a simple query that, when it is executed, will produce a list of all students whose score on the first test was greater than 90. Note that because the whole `Student` object is selected, the type of the query is `IEnumerable<Student>`. Although the code could also use implicit typing by using the `var` keyword, explicit typing is used to clearly illustrate results. (For more information about `var`, see [Implicitly Typed Local Variables](#).)

Note also that the query's range variable, `student`, serves as a reference to each `Student` in the source, providing member access for each object.

```

// Create the query.
// The first line could also be written as "var studentQuery ="
IEnumerable<Student> studentQuery =
    from student in students
    where student.Scores[0] > 90
    select student;

```

Execute the Query

To execute the query

1. Now write the `foreach` loop that will cause the query to execute. Note the following about the code:
 - Each element in the returned sequence is accessed through the iteration variable in the `foreach` loop.
 - The type of this variable is `Student`, and the type of the query variable is compatible,

```
IEnumerable<Student> .
```

2. After you have added this code, build and run the application to see the results in the **Console** window.

```
// Execute the query.  
// var could be used here also.  
foreach (Student student in studentQuery)  
{  
    Console.WriteLine("{0}, {1}", student.Last, student.First);  
}  
  
// Output:  
// Omelchenko, Svetlana  
// Garcia, Cesar  
// Fakhouri, Fadi  
// Feng, Hanying  
// Garcia, Hugo  
// Adams, Terry  
// Zabokritski, Eugene  
// Tucker, Michael
```

To add another filter condition

1. You can combine multiple Boolean conditions in the `where` clause in order to further refine a query. The following code adds a condition so that the query returns those students whose first score was over 90 and whose last score was less than 80. The `where` clause should resemble the following code.

```
where student.Scores[0] > 90 && student.Scores[3] < 80
```

For more information, see [where clause](#).

Modify the Query

To order the results

1. It will be easier to scan the results if they are in some kind of order. You can order the returned sequence by any accessible field in the source elements. For example, the following `orderby` clause orders the results in alphabetical order from A to Z according to the last name of each student. Add the following `orderby` clause to your query, right after the `where` statement and before the `select` statement:

```
orderby student.Last ascending
```

2. Now change the `orderby` clause so that it orders the results in reverse order according to the score on the first test, from the highest score to the lowest score.

```
orderby student.Scores[0] descending
```

3. Change the `WriteLine` format string so that you can see the scores:

```
Console.WriteLine("{0}, {1} {2}", student.Last, student.First, student.Scores[0]);
```

For more information, see [orderby clause](#).

To group the results

1. Grouping is a powerful capability in query expressions. A query with a group clause produces a sequence of groups, and each group itself contains a `key` and a sequence that consists of all the members of that group. The following new query groups the students by using the first letter of their last name as the key.

```
// studentQuery2 is an IEnumerable<IGrouping<char, Student>>
var studentQuery2 =
    from student in students
    group student by student.Last[0];
```

- Note that the type of the query has now changed. It now produces a sequence of groups that have a `char` type as a key, and a sequence of `Student` objects. Because the type of the query has changed, the following code changes the `foreach` execution loop also:

```
// studentGroup is a IGrouping<char, Student>
foreach (var studentGroup in studentQuery2)
{
    Console.WriteLine(studentGroup.Key);
    foreach (Student student in studentGroup)
    {
        Console.WriteLine("    {0}, {1}",
                           student.Last, student.First);
    }
}

// Output:
// O
//  Omelchenko, Svetlana
//  O'Donnell, Claire
// M
//  Mortensen, Sven
// G
//  Garcia, Cesar
//  Garcia, Debra
//  Garcia, Hugo
// F
//  Fakhouri, Fadi
//  Feng, Hanying
// T
//  Tucker, Lance
//  Tucker, Michael
// A
//  Adams, Terry
// Z
//  Zabokritski, Eugene
```

- Run the application and view the results in the **Console** window.

For more information, see [group clause](#).

To make the variables implicitly typed

- Explicitly coding `IEnumerables` of `IGroupings` can quickly become tedious. You can write the same query and `foreach` loop much more conveniently by using `var`. The `var` keyword does not change the types of your objects; it just instructs the compiler to infer the types. Change the type of `studentQuery` and the iteration variable `group` to `var` and rerun the query. Note that in the inner `foreach` loop, the iteration variable is still typed as `Student`, and the query works just as before. Change the `student` iteration variable to `var` and run the query again. You see that you get exactly the same results.

```

var studentQuery3 =
    from student in students
    group student by student.Last[0];

foreach (var groupOfStudents in studentQuery3)
{
    Console.WriteLine(groupOfStudents.Key);
    foreach (var student in groupOfStudents)
    {
        Console.WriteLine("    {0}, {1}",
            student.Last, student.First);
    }
}

// Output:
// O
//   Omelchenko, Svetlana
//   O'Donnell, Claire
// M
//   Mortensen, Sven
// G
//   Garcia, Cesar
//   Garcia, Debra
//   Garcia, Hugo
// F
//   Fakhouri, Fadi
//   Feng, Hanying
// T
//   Tucker, Lance
//   Tucker, Michael
// A
//   Adams, Terry
// Z
//   Zabokritski, Eugene

```

For more information about [var](#), see [Implicitly Typed Local Variables](#).

To order the groups by their key value

1. When you run the previous query, you notice that the groups are not in alphabetical order. To change this, you must provide an `orderby` clause after the `group` clause. But to use an `orderby` clause, you first need an identifier that serves as a reference to the groups created by the `group` clause. You provide the identifier by using the `into` keyword, as follows:


```

var studentQuery4 =
    from student in students
    group student by student.Last[0] into studentGroup
    orderby studentGroup.Key
    select studentGroup;

foreach (var groupOfStudents in studentQuery4)
{
    Console.WriteLine(groupOfStudents.Key);
    foreach (var student in groupOfStudents)
    {
        Console.WriteLine("    {0}, {1}",
            student.Last, student.First);
    }
}

// Output:
//A
//  Adams, Terry
//F
//  Fakhouri, Fadi
//  Feng, Hanying
//G
//  Garcia, Cesar
//  Garcia, Debra
//  Garcia, Hugo
//M
//  Mortensen, Sven
//O
//  Omelchenko, Svetlana
//  O'Donnell, Claire
//T
//  Tucker, Lance
//  Tucker, Michael
//Z
//  Zabokritski, Eugene

```

When you run this query, you will see the groups are now sorted in alphabetical order.

To introduce an identifier by using let

1. You can use the `let` keyword to introduce an identifier for any expression result in the query expression. This identifier can be a convenience, as in the following example, or it can enhance performance by storing the results of an expression so that it does not have to be calculated multiple times.

```
// studentQuery5 is an IEnumerable<string>
// This query returns those students whose
// first test score was higher than their
// average score.
var studentQuery5 =
    from student in students
    let totalScore = student.Scores[0] + student.Scores[1] +
        student.Scores[2] + student.Scores[3]
    where totalScore / 4 < student.Scores[0]
    select student.Last + " " + student.First;

foreach (string s in studentQuery5)
{
    Console.WriteLine(s);
}

// Output:
// Omelchenko Svetlana
// O'Donnell Claire
// Mortensen Sven
// Garcia Cesar
// Fakhouri Fadi
// Feng Hanying
// Garcia Hugo
// Adams Terry
// Zabokritski Eugene
// Tucker Michael
```

For more information, see [let clause](#).

To use method syntax in a query expression

1. As described in [Query Syntax and Method Syntax in LINQ](#), some query operations can only be expressed by using method syntax. The following code calculates the total score for each `Student` in the source sequence, and then calls the `Average()` method on the results of that query to calculate the average score of the class.

```
var studentQuery6 =
    from student in students
    let totalScore = student.Scores[0] + student.Scores[1] +
        student.Scores[2] + student.Scores[3]
    select totalScore;

double averageScore = studentQuery6.Average();
Console.WriteLine("Class average score = {0}", averageScore);

// Output:
// Class average score = 334.166666666667
```

To transform or project in the select clause

1. It is very common for a query to produce a sequence whose elements differ from the elements in the source sequences. Delete or comment out your previous query and execution loop, and replace it with the following code. Note that the query returns a sequence of strings (not `Students`), and this fact is reflected in the `foreach` loop.

```

IEnumerable<string> studentQuery7 =
    from student in students
    where student.Last == "Garcia"
    select student.First;

Console.WriteLine("The Garcias in the class are:");
foreach (string s in studentQuery7)
{
    Console.WriteLine(s);
}

// Output:
// The Garcias in the class are:
// Cesar
// Debra
// Hugo

```

2. Code earlier in this walkthrough indicated that the average class score is approximately 334. To produce a sequence of `students` whose total score is greater than the class average, together with their `Student ID`, you can use an anonymous type in the `select` statement:

```

var studentQuery8 =
    from student in students
    let x = student.Scores[0] + student.Scores[1] +
        student.Scores[2] + student.Scores[3]
    where x > averageScore
    select new { id = student.ID, score = x };

foreach (var item in studentQuery8)
{
    Console.WriteLine("Student ID: {0}, Score: {1}", item.id, item.score);
}

// Output:
// Student ID: 113, Score: 338
// Student ID: 114, Score: 353
// Student ID: 116, Score: 369
// Student ID: 117, Score: 352
// Student ID: 118, Score: 343
// Student ID: 120, Score: 341
// Student ID: 122, Score: 368

```

Next Steps

After you are familiar with the basic aspects of working with queries in C#, you are ready to read the documentation and samples for the specific type of LINQ provider you are interested in:

[LINQ to SQL](#)

[LINQ to DataSet](#)

[LINQ to XML \(C#\)](#)

[LINQ to Objects \(C#\)](#)

See also

- [Language-Integrated Query \(LINQ\) \(C#\)](#)
- [LINQ Query Expressions](#)

Standard Query Operators Overview (C#)

12/28/2021 • 3 minutes to read • [Edit Online](#)

The *standard query operators* are the methods that form the LINQ pattern. Most of these methods operate on sequences, where a sequence is an object whose type implements the [IEnumerable<T>](#) interface or the [IQueryable<T>](#) interface. The standard query operators provide query capabilities including filtering, projection, aggregation, sorting and more.

There are two sets of LINQ standard query operators: one that operates on objects of type [IEnumerable<T>](#), another that operates on objects of type [IQueryable<T>](#). The methods that make up each set are static members of the [Enumerable](#) and [Queryable](#) classes, respectively. They are defined as *extension methods* of the type that they operate on. Extension methods can be called by using either static method syntax or instance method syntax.

In addition, several standard query operator methods operate on types other than those based on [IEnumerable<T>](#) or [IQueryable<T>](#). The [Enumerable](#) type defines two such methods that both operate on objects of type [IEnumerable](#). These methods, [Cast<TResult>\(IEnumerable\)](#) and [OfType<TResult>\(IEnumerable\)](#), let you enable a non-parameterized, or non-generic, collection to be queried in the LINQ pattern. They do this by creating a strongly typed collection of objects. The [Queryable](#) class defines two similar methods, [Cast<TResult>\(IQueryable\)](#) and [OfType<TResult>\(IQueryable\)](#), that operate on objects of type [IQueryable](#).

The standard query operators differ in the timing of their execution, depending on whether they return a singleton value or a sequence of values. Those methods that return a singleton value (for example, [Average](#) and [Sum](#)) execute immediately. Methods that return a sequence defer the query execution and return an enumerable object.

For methods that operate on in-memory collections, that is, those methods that extend [IEnumerable<T>](#), the returned enumerable object captures the arguments that were passed to the method. When that object is enumerated, the logic of the query operator is employed and the query results are returned.

In contrast, methods that extend [IQueryable<T>](#) don't implement any querying behavior. They build an expression tree that represents the query to be performed. The query processing is handled by the source [IQueryable<T>](#) object.

Calls to query methods can be chained together in one query, which enables queries to become arbitrarily complex.

The following code example demonstrates how the standard query operators can be used to obtain information about a sequence.

```

string sentence = "the quick brown fox jumps over the lazy dog";
// Split the string into individual words to create a collection.
string[] words = sentence.Split(' ');

// Using query expression syntax.
var query = from word in words
            group word.ToUpper() by word.Length into gr
            orderby gr.Key
            select new { Length = gr.Key, Words = gr };

// Using method-based query syntax.
var query2 = words.
    GroupBy(w => w.Length, w => w.ToUpper()).
    Select(g => new { Length = g.Key, Words = g }).
    OrderBy(o => o.Length);

foreach (var obj in query)
{
    Console.WriteLine("Words of length {0}:", obj.Length);
    foreach (string word in obj.Words)
        Console.WriteLine(word);
}

// This code example produces the following output:
//
// Words of length 3:
// THE
// FOX
// THE
// DOG
// Words of length 4:
// OVER
// LAZY
// Words of length 5:
// QUICK
// BROWN
// JUMPS

```

Query Expression Syntax

Some of the more frequently used standard query operators have dedicated C# and Visual Basic language keyword syntax that enables them to be called as part of a *query expression*. For more information about standard query operators that have dedicated keywords and their corresponding syntaxes, see [Query Expression Syntax for Standard Query Operators \(C#\)](#).

Extending the Standard Query Operators

You can augment the set of standard query operators by creating domain-specific methods that are appropriate for your target domain or technology. You can also replace the standard query operators with your own implementations that provide additional services such as remote evaluation, query translation, and optimization. See [AsEnumerable](#) for an example.

Related Sections

The following links take you to articles that provide additional information about the various standard query operators based on functionality.

[Sorting Data \(C#\)](#)

[Set Operations \(C#\)](#)

[Filtering Data \(C#\)](#)

[Quantifier Operations \(C#\)](#)

[Projection Operations \(C#\)](#)

[Partitioning Data \(C#\)](#)

[Join Operations \(C#\)](#)

[Grouping Data \(C#\)](#)

[Generation Operations \(C#\)](#)

[Equality Operations \(C#\)](#)

[Element Operations \(C#\)](#)

[Converting Data Types \(C#\)](#)

[Concatenation Operations \(C#\)](#)

[Aggregation Operations \(C#\)](#)

See also

- [Enumerable](#)
- [Queryable](#)
- [Introduction to LINQ Queries \(C#\)](#)
- [Query Expression Syntax for Standard Query Operators \(C#\)](#)
- [Classification of Standard Query Operators by Manner of Execution \(C#\)](#)
- [Extension Methods](#)

Query Expression Syntax for Standard Query Operators (C#)

12/28/2021 • 2 minutes to read • [Edit Online](#)

Some of the more frequently used standard query operators have dedicated C# language keyword syntax that enables them to be called as part of a *query expression*. A query expression is a different, more readable form of expressing a query than its *method-based* equivalent. Query expression clauses are translated into calls to the query methods at compile time.

Query Expression Syntax Table

The following table lists the standard query operators that have equivalent query expression clauses.

METHOD	C# QUERY EXPRESSION SYNTAX
Cast	Use an explicitly typed range variable, for example: <code>from int i in numbers</code> (For more information, see from clause .)
GroupBy	<code>group ... by</code> -or- <code>group ... by ... into ...</code> (For more information, see group clause .)
GroupJoin<TOuter,TInner,TKey,TResult>(IEnumerable<TOuter>, IEnumerable<TInner>, Func<TOuter,TKey>, Func<TInner,TKey>, Func<TOuter,IEnumerable<TInner>,TResult>)	<code>join ... in ... on ... equals ... into ...</code> (For more information, see join clause .)
Join<TOuter,TInner,TKey,TResult>(IEnumerable<TOuter>, IEnumerable<TInner>, Func<TOuter,TKey>, Func<TInner,TKey>, Func<TOuter,TInner,TResult>)	<code>join ... in ... on ... equals ...</code> (For more information, see join clause .)
OrderBy<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>)	<code>orderby</code> (For more information, see orderby clause .)
OrderByDescending<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>)	<code>orderby ... descending</code> (For more information, see orderby clause .)
Select	<code>select</code> (For more information, see select clause .)

METHOD	C# QUERY EXPRESSION SYNTAX
SelectMany	Multiple <code>from</code> clauses. (For more information, see from clause .)
ThenBy<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>)	<code>orderby ..., ...</code> (For more information, see orderby clause .)
ThenByDescending<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>)	<code>orderby ..., ... descending</code> (For more information, see orderby clause .)
Where	<code>where</code> (For more information, see where clause .)

See also

- [Enumerable](#)
- [Queryable](#)
- [Standard Query Operators Overview \(C#\)](#)
- [Classification of Standard Query Operators by Manner of Execution \(C#\)](#)

Classification of Standard Query Operators by Manner of Execution (C#)

12/28/2021 • 3 minutes to read • [Edit Online](#)

The LINQ to Objects implementations of the standard query operator methods execute in one of two main ways: immediate or deferred. The query operators that use deferred execution can be additionally divided into two categories: streaming and non-streaming. If you know how the different query operators execute, it may help you understand the results that you get from a given query. This is especially true if the data source is changing or if you are building a query on top of another query. This topic classifies the standard query operators according to their manner of execution.

Manners of Execution

Immediate

Immediate execution means that the data source is read and the operation is performed at the point in the code where the query is declared. All the standard query operators that return a single, non-enumerable result execute immediately.

Deferred

Deferred execution means that the operation is not performed at the point in the code where the query is declared. The operation is performed only when the query variable is enumerated, for example by using a `foreach` statement. This means that the results of executing the query depend on the contents of the data source when the query is executed rather than when the query is defined. If the query variable is enumerated multiple times, the results might differ every time. Almost all the standard query operators whose return type is `IEnumerable<T>` or `IOrderedEnumerable<TElement>` execute in a deferred manner.

Query operators that use deferred execution can be additionally classified as streaming or non-streaming.

Streaming

Streaming operators do not have to read all the source data before they yield elements. At the time of execution, a streaming operator performs its operation on each source element as it is read and yields the element if appropriate. A streaming operator continues to read source elements until a result element can be produced. This means that more than one source element might be read to produce one result element.

Non-Streaming

Non-streaming operators must read all the source data before they can yield a result element. Operations such as sorting or grouping fall into this category. At the time of execution, non-streaming query operators read all the source data, put it into a data structure, perform the operation, and yield the resulting elements.

Classification Table

The following table classifies each standard query operator method according to its method of execution.

NOTE

If an operator is marked in two columns, two input sequences are involved in the operation, and each sequence is evaluated differently. In these cases, it is always the first sequence in the parameter list that is evaluated in a deferred, streaming manner.

STANDARD QUERY OPERATOR	RETURN TYPE	IMMEDIATE EXECUTION	DEFERRED STREAMING EXECUTION	DEFERRED NON-STREAMING EXECUTION
Aggregate	TSource	X		
All	Boolean	X		
Any	Boolean	X		
AsEnumerable	IEnumerable<T>		X	
Average	Single numeric value	X		
Cast	IEnumerable<T>		X	
Concat	IEnumerable<T>		X	
Contains	Boolean	X		
Count	Int32	X		
DefaultIfEmpty	IEnumerable<T>		X	
Distinct	IEnumerable<T>		X	
ElementAt	TSource	X		
ElementAtOrDefault	TSource	X		
Empty	IEnumerable<T>	X		
Except	IEnumerable<T>		X	X
First	TSource	X		
FirstOrDefault	TSource	X		
GroupBy	IEnumerable<T>			X
GroupJoin	IEnumerable<T>		X	X
Intersect	IEnumerable<T>		X	X
Join	IEnumerable<T>		X	X
Last	TSource	X		
LastOrDefault	TSource	X		
LongCount	Int64	X		

STANDARD QUERY OPERATOR	RETURN TYPE	IMMEDIATE EXECUTION	DEFERRED STREAMING EXECUTION	DEFERRED NON-STREAMING EXECUTION
Max	Single numeric value, TSource, or TResult	X		
Min	Single numeric value, TSource, or TResult	X		
OfType	IEnumerable<T>		X	
OrderBy	IOrderedEnumerable<TElement>			X
OrderByDescending	IOrderedEnumerable<TElement>			X
Range	IEnumerable<T>		X	
Repeat	IEnumerable<T>		X	
Reverse	IEnumerable<T>			X
Select	IEnumerable<T>		X	
SelectMany	IEnumerable<T>		X	
SequenceEqual	Boolean	X		
Single	TSource	X		
SingleOrDefault	TSource	X		
Skip	IEnumerable<T>		X	
SkipWhile	IEnumerable<T>		X	
Sum	Single numeric value	X		
Take	IEnumerable<T>		X	
TakeWhile	IEnumerable<T>		X	
ThenBy	IOrderedEnumerable<TElement>			X
ThenByDescending	IOrderedEnumerable<TElement>			X
ToArray	TSource array	X		
ToDictionary	Dictionary<TKey,TValue>	X		

STANDARD QUERY OPERATOR	RETURN TYPE	IMMEDIATE EXECUTION	DEFERRED STREAMING EXECUTION	DEFERRED NON-STREAMING EXECUTION
ToList	IList<T>	X		
ToLookup	ILookup<TKey,TElement>	X		
Union	IEnumerable<T>		X	
Where	IEnumerable<T>		X	

See also

- [Enumerable](#)
- [Standard Query Operators Overview \(C#\)](#)
- [Query Expression Syntax for Standard Query Operators \(C#\)](#)
- [LINQ to Objects \(C#\)](#)

Sorting Data (C#)

12/28/2021 • 2 minutes to read • [Edit Online](#)

A sorting operation orders the elements of a sequence based on one or more attributes. The first sort criterion performs a primary sort on the elements. By specifying a second sort criterion, you can sort the elements within each primary sort group.

The following illustration shows the results of an alphabetical sort operation on a sequence of characters:

Source

G C F E B A D

Results

A B C D E F G

The standard query operator methods that sort data are listed in the following section.

Methods

METHOD NAME	DESCRIPTION	C# QUERY EXPRESSION SYNTAX	MORE INFORMATION
OrderBy	Sorts values in ascending order.	<code>orderby</code>	Enumerable.OrderBy Queryable.OrderBy
OrderByDescending	Sorts values in descending order.	<code>orderby ... descending</code>	Enumerable.OrderByDescending Queryable.OrderByDescending
ThenBy	Performs a secondary sort in ascending order.	<code>orderby ..., ...</code>	Enumerable.ThenBy Queryable.ThenBy
ThenByDescending	Performs a secondary sort in descending order.	<code>orderby ..., ... descending</code>	Enumerable.ThenByDescending Queryable.ThenByDescending
Reverse	Reverses the order of the elements in a collection.	Not applicable.	Enumerable.Reverse Queryable.Reverse

Query Expression Syntax Examples

Primary Sort Examples

Primary Ascending Sort

The following example demonstrates how to use the `orderby` clause in a LINQ query to sort the strings in an array by string length, in ascending order.

```

string[] words = { "the", "quick", "brown", "fox", "jumps" };

IEnumerable<string> query = from word in words
                           orderby word.Length
                           select word;

foreach (string str in query)
    Console.WriteLine(str);

/* This code produces the following output:

    the
    fox
    quick
    brown
    jumps
*/

```

Primary Descending Sort

The next example demonstrates how to use the `orderby descending` clause in a LINQ query to sort the strings by their first letter, in descending order.

```

string[] words = { "the", "quick", "brown", "fox", "jumps" };

IEnumerable<string> query = from word in words
                           orderby word.Substring(0, 1) descending
                           select word;

foreach (string str in query)
    Console.WriteLine(str);

/* This code produces the following output:

    the
    quick
    jumps
    fox
    brown
*/

```

Secondary Sort Examples

Secondary Ascending Sort

The following example demonstrates how to use the `orderby` clause in a LINQ query to perform a primary and secondary sort of the strings in an array. The strings are sorted primarily by length and secondarily by the first letter of the string, both in ascending order.

```

string[] words = { "the", "quick", "brown", "fox", "jumps" };

IEnumerable<string> query = from word in words
                           orderby word.Length, word.Substring(0, 1)
                           select word;

foreach (string str in query)
    Console.WriteLine(str);

/* This code produces the following output:

    fox
    the
    brown
    jumps
    quick
*/

```

Secondary Descending Sort

The next example demonstrates how to use the `orderby descending` clause in a LINQ query to perform a primary sort, in ascending order, and a secondary sort, in descending order. The strings are sorted primarily by length and secondarily by the first letter of the string.

```

string[] words = { "the", "quick", "brown", "fox", "jumps" };

IEnumerable<string> query = from word in words
                           orderby word.Length, word.Substring(0, 1) descending
                           select word;

foreach (string str in query)
    Console.WriteLine(str);

/* This code produces the following output:

    the
    fox
    quick
    jumps
    brown
*/

```

See also

- [System.Linq](#)
- [Standard Query Operators Overview \(C#\)](#)
- [orderby clause](#)
- [Order the results of a join clause](#)
- [How to sort or filter text data by any word or field \(LINQ\) \(C#\)](#)

Set operations (C#)

12/28/2021 • 7 minutes to read • [Edit Online](#)

Set operations in LINQ refer to query operations that produce a result set that is based on the presence or absence of equivalent elements within the same or separate collections (or sets).

The standard query operator methods that perform set operations are listed in the following section.

Methods

METHOD NAMES	DESCRIPTION	C# QUERY EXPRESSION SYNTAX	MORE INFORMATION
Distinct or DistinctBy	Removes duplicate values from a collection.	Not applicable.	Enumerable.Distinct Enumerable.DistinctBy Queryable.Distinct Queryable.DistinctBy
Except or ExceptBy	Returns the set difference, which means the elements of one collection that do not appear in a second collection.	Not applicable.	Enumerable.Except Enumerable.ExceptBy Queryable.Except Queryable.ExceptBy
Intersect or IntersectBy	Returns the set intersection, which means elements that appear in each of two collections.	Not applicable.	Enumerable.Intersect Enumerable.IntersectBy Queryable.Intersect Queryable.IntersectBy
Union or UnionBy	Returns the set union, which means unique elements that appear in either of two collections.	Not applicable.	Enumerable.Union Enumerable.UnionBy Queryable.Union Queryable.UnionBy

Examples

Some of the following examples rely on a `record` type that represents the planets in our solar system.


```

namespace SolarSystem;

record Planet(
    string Name,
    PlanetType Type,
    int OrderFromSun)
{
    public static readonly Planet Mercury =
        new(nameof(Mercury), PlanetType.Rock, 1);

    public static readonly Planet Venus =
        new(nameof(Venus), PlanetType.Rock, 2);

    public static readonly Planet Earth =
        new(nameof(Earth), PlanetType.Rock, 3);

    public static readonly Planet Mars =
        new(nameof(Mars), PlanetType.Rock, 4);

    public static readonly Planet Jupiter =
        new(nameof(Jupiter), PlanetType.Gas, 5);

    public static readonly Planet Saturn =
        new(nameof(Saturn), PlanetType.Gas, 6);

    public static readonly Planet Uranus =
        new(nameof(Uranus), PlanetType.Liquid, 7);

    public static readonly Planet Neptune =
        new(nameof(Neptune), PlanetType.Liquid, 8);

    // Yes, I know... not technically a planet anymore
    public static readonly Planet Pluto =
        new(nameof(Pluto), PlanetType.Ice, 9);
}

```

The `record Planet` is a positional record, which requires a `Name`, `Type`, and `OrderFromSun` arguments to instantiate it. There are several `static readonly` planet instances on the `Planet` type. These are convenience-based definitions for well-known planets. The `Type` member identifies the planet type.

```

namespace SolarSystem;

enum PlanetType
{
    Rock,
    Ice,
    Gas,
    Liquid
};

```

Distinct and DistinctBy

The following example depicts the behavior of the [Enumerable.Distinct](#) method on a sequence of strings. The returned sequence contains the unique elements from the input sequence.



```

string[] planets = { "Mercury", "Venus", "Venus", "Earth", "Mars", "Earth" };

IEnumerable<string> query = from planet in planets.Distinct()
                           select planet;

foreach (var str in query)
{
    Console.WriteLine(str);
}

/* This code produces the following output:
 *
 * Mercury
 * Venus
 * Earth
 * Mars
 */

```

The `DistinctBy` is an alternative approach to `Distinct` that takes a `keySelector`. The `keySelector` is used as the comparative discriminator of the source type. Consider the following planet array:

```

Planet[] planets =
{
    Planet.Mercury,
    Planet.Venus,
    Planet.Earth,
    Planet.Mars,
    Planet.Jupiter,
    Planet.Saturn,
    Planet.Uranus,
    Planet.Neptune,
    Planet.Pluto
};

```

In the following code, planets are discriminated based on their `PlanetType`, and the first planet of each type is displayed:

```

foreach (Planet planet in planets.DistinctBy(p => p.Type))
{
    Console.WriteLine(planet);
}

// This code produces the following output:
// Planet { Name = Mercury, Type = Rock, OrderFromSun = 1 }
// Planet { Name = Jupiter, Type = Gas, OrderFromSun = 5 }
// Planet { Name = Uranus, Type = Liquid, OrderFromSun = 7 }
// Planet { Name = Pluto, Type = Ice, OrderFromSun = 9 }

```

In the preceding C# code:

- The `Planet` array is filtered distinctly to the first occurrence of each unique planet type.
- The resulting `planet` instances are written to the console.

Except and ExceptBy

The following example depicts the behavior of `Enumerable.Except`. The returned sequence contains only the elements from the first input sequence that are not in the second input sequence.



```
string[] planets1 = { "Mercury", "Venus", "Earth", "Jupiter" };
string[] planets2 = { "Mercury", "Earth", "Mars", "Jupiter" };

IEnumerable<string> query = from planet in planets1.Except(planets2)
                           select planet;

foreach (var str in query)
{
    Console.WriteLine(str);
}

/* This code produces the following output:
 *
 * Venus
 */
```

The `ExceptBy` method is an alternative approach to `Except` that takes two sequences of possibly heterogeneous types and a `keySelector`. The `keySelector` is the same type as the second collection's type, and it is used as the comparative discriminator of the source type. Consider the following planet arrays:

```
Planet[] planets =
{
    Planet.Mercury,
    Planet.Venus,
    Planet.Earth,
    Planet.Jupiter
};

Planet[] morePlanets =
{
    Planet.Mercury,
    Planet.Earth,
    Planet.Mars,
    Planet.Jupiter
};
```

To find planets in the first collection that aren't in the second collection, you can project the planet names as the `second` collection and provide the same `keySelector`:

```
// A shared "keySelector"
static string PlanetNameSelector(Planet planet) => planet.Name;

foreach (Planet planet in
    planets.ExceptBy(
        morePlanets.Select(PlanetNameSelector), PlanetNameSelector))
{
    Console.WriteLine(planet);
}

// This code produces the following output:
// Planet { Name = Venus, Type = Rock, OrderFromSun = 2 }
```

In the preceding C# code:

- The `keySelector` is defined as a `static` local function that discriminates on a planet name.
- The first planet array is filtered to planets that are not found in the second planet array, based on their name.

- The resulting `planet` instance is written to the console.

Intersect and IntersectBy

The following example depicts the behavior of [Enumerable.Intersect](#). The returned sequence contains the elements that are common to both of the input sequences.



```
string[] planets1 = { "Mercury", "Venus", "Earth", "Jupiter" };
string[] planets2 = { "Mercury", "Earth", "Mars", "Jupiter" };

IEnumerable<string> query = from planet in planets1.Intersect(planets2)
                           select planet;

foreach (var str in query)
{
    Console.WriteLine(str);
}

/* This code produces the following output:
 *
 * Mercury
 * Earth
 * Jupiter
 */
```

The [IntersectBy](#) method is an alternative approach to `Intersect` that takes two sequences of possibly heterogeneous types and a `keySelector`. The `keySelector` is used as the comparative discriminator of the second collection's type. Consider the following planet arrays:

```
Planet[] firstFivePlanetsFromTheSun =
{
    Planet.Mercury,
    Planet.Venus,
    Planet.Earth,
    Planet.Mars,
    Planet.Jupiter
};

Planet[] lastFivePlanetsFromTheSun =
{
    Planet.Mars,
    Planet.Jupiter,
    Planet.Saturn,
    Planet.Uranus,
    Planet.Neptune
};
```

There are two arrays of planets; one represents the first five planets from the sun and the second represents the last five planets from the sun. Since the `Planet` type is a positional `record` type, you can use its value comparison semantics in the form of the `keySelector`:

```
foreach (Planet planet in
    firstFivePlanetsFromTheSun.IntersectBy(
        lastFivePlanetsFromTheSun, planet => planet))
{
    Console.WriteLine(planet);
}

// This code produces the following output:
//     Planet { Name = Mars, Type = Rock, OrderFromSun = 4 }
//     Planet { Name = Jupiter, Type = Gas, OrderFromSun = 5 }
```

In the preceding C# code:

- The two `Planet` arrays are intersected by their value comparison semantics.
- Only planets that are found in both arrays are present in the resulting sequence.
- The resulting `planet` instances are written to the console.

Union and UnionBy

The following example depicts a union operation on two sequences of strings. The returned sequence contains the unique elements from both input sequences.



```
string[] planets1 = { "Mercury", "Venus", "Earth", "Jupiter" };
string[] planets2 = { "Mercury", "Earth", "Mars", "Jupiter" };

IEnumerable<string> query = from planet in planets1.Union(planets2)
                           select planet;

foreach (var str in query)
{
    Console.WriteLine(str);
}

/* This code produces the following output:
*
* Mercury
* Venus
* Earth
* Jupiter
* Mars
*/
```

The `UnionBy` method is an alternative approach to `Union` that takes two sequences of the same type and a `keySelector`. The `keySelector` is used as the comparative discriminator of the source type. Consider the following planet arrays:

```
Planet[] firstFivePlanetsFromTheSun =
{
    Planet.Mercury,
    Planet.Venus,
    Planet.Earth,
    Planet.Mars,
    Planet.Jupiter
};

Planet[] lastFivePlanetsFromTheSun =
{
    Planet.Mars,
    Planet.Jupiter,
    Planet.Saturn,
    Planet.Uranus,
    Planet.Neptune
};
```

To union these two collections into a single sequence, you provide the `keySelector`:

```
foreach (Planet planet in
    firstFivePlanetsFromTheSun.UnionBy(
        lastFivePlanetsFromTheSun, planet => planet))
{
    Console.WriteLine(planet);
}

// This code produces the following output:
// Planet { Name = Mercury, Type = Rock, OrderFromSun = 1 }
// Planet { Name = Venus, Type = Rock, OrderFromSun = 2 }
// Planet { Name = Earth, Type = Rock, OrderFromSun = 3 }
// Planet { Name = Mars, Type = Rock, OrderFromSun = 4 }
// Planet { Name = Jupiter, Type = Gas, OrderFromSun = 5 }
// Planet { Name = Saturn, Type = Gas, OrderFromSun = 6 }
// Planet { Name = Uranus, Type = Liquid, OrderFromSun = 7 }
// Planet { Name = Neptune, Type = Liquid, OrderFromSun = 8 }
```

In the preceding C# code:

- The two `Planet` arrays are weaved together using their `record` value comparison semantics.
- The resulting `planet` instances are written to the console.

See also

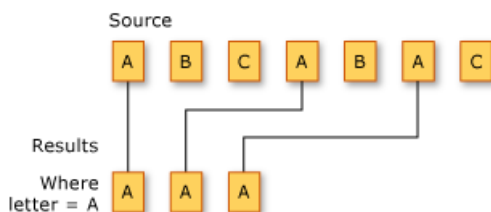
- [System.Linq](#)
- [Standard Query Operators Overview \(C#\)](#)
- [How to combine and compare string collections \(LINQ\) \(C#\)](#)
- [How to find the set difference between two lists \(LINQ\) \(C#\)](#)

Filtering Data (C#)

12/28/2021 • 2 minutes to read • [Edit Online](#)

Filtering refers to the operation of restricting the result set to contain only those elements that satisfy a specified condition. It is also known as selection.

The following illustration shows the results of filtering a sequence of characters. The predicate for the filtering operation specifies that the character must be 'A'.



The standard query operator methods that perform selection are listed in the following section.

Methods

METHOD NAME	DESCRIPTION	C# QUERY EXPRESSION SYNTAX	MORE INFORMATION
OfType	Selects values, depending on their ability to be cast to a specified type.	Not applicable.	Enumerable.OfType Queryable.OfType
Where	Selects values that are based on a predicate function.	<code>where</code>	Enumerable.Where Queryable.Where

Query Expression Syntax Example

The following example uses the `where` clause to filter from an array those strings that have a specific length.

```
string[] words = { "the", "quick", "brown", "fox", "jumps" };

IEnumerable<string> query = from word in words
    where word.Length == 3
    select word;

foreach (string str in query)
    Console.WriteLine(str);

/* This code produces the following output:

    the
    fox
*/
```

See also

- [System.Linq](#)

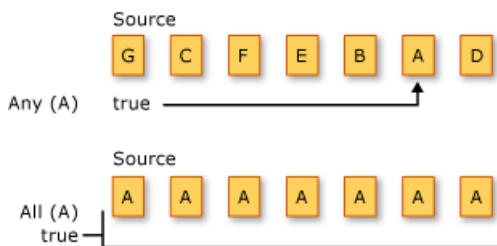
- [Standard Query Operators Overview \(C#\)](#)
- [where clause](#)
- [Dynamically specify predicate filters at run time](#)
- [How to query an assembly's metadata with Reflection \(LINQ\) \(C#\)](#)
- [How to query for files with a specified attribute or name \(C#\)](#)
- [How to sort or filter text data by any word or field \(LINQ\) \(C#\)](#)

Quantifier Operations (C#)

12/28/2021 • 2 minutes to read • [Edit Online](#)

Quantifier operations return a [Boolean](#) value that indicates whether some or all of the elements in a sequence satisfy a condition.

The following illustration depicts two different quantifier operations on two different source sequences. The first operation asks if one or more of the elements are the character 'A', and the result is `true`. The second operation asks if all the elements are the character 'A', and the result is `true`.



The standard query operator methods that perform quantifier operations are listed in the following section.

Methods

METHOD NAME	DESCRIPTION	C# QUERY EXPRESSION SYNTAX	MORE INFORMATION
All	Determines whether all the elements in a sequence satisfy a condition.	Not applicable.	Enumerable.All Queryable.All
Any	Determines whether any elements in a sequence satisfy a condition.	Not applicable.	Enumerable.Any Queryable.Any
Contains	Determines whether a sequence contains a specified element.	Not applicable.	Enumerable.Contains Queryable.Contains

Query Expression Syntax Examples

All

The following example uses the `All` to check that all strings are of a specific length.

```

class Market
{
    public string Name { get; set; }
    public string[] Items { get; set; }
}

public static void Example()
{
    List<Market> markets = new List<Market>
    {
        new Market { Name = "Emily's", Items = new string[] { "kiwi", "cheery", "banana" } },
        new Market { Name = "Kim's", Items = new string[] { "melon", "mango", "olive" } },
        new Market { Name = "Adam's", Items = new string[] { "kiwi", "apple", "orange" } },
    };

    // Determine which market have all fruit names length equal to 5
    IEnumerable<string> names = from market in markets
                               where market.Items.All(item => item.Length == 5)
                               select market.Name;

    foreach (string name in names)
    {
        Console.WriteLine($"{name} market");
    }

    // This code produces the following output:
    //
    // Kim's market
}

```

Any

The following example uses the `Any` to check that any strings are start with 'o'.

```

class Market
{
    public string Name { get; set; }
    public string[] Items { get; set; }
}

public static void Example()
{
    List<Market> markets = new List<Market>
    {
        new Market { Name = "Emily's", Items = new string[] { "kiwi", "cheery", "banana" } },
        new Market { Name = "Kim's", Items = new string[] { "melon", "mango", "olive" } },
        new Market { Name = "Adam's", Items = new string[] { "kiwi", "apple", "orange" } },
    };

    // Determine which market have any fruit names start with 'o'
    IEnumerable<string> names = from market in markets
                               where market.Items.Any(item => item.StartsWith("o"))
                               select market.Name;

    foreach (string name in names)
    {
        Console.WriteLine($"{name} market");
    }

    // This code produces the following output:
    //
    // Kim's market
    // Adam's market
}

```

Contains

The following example uses the `Contains` to check an array have a specific element.

```
class Market
{
    public string Name { get; set; }
    public string[] Items { get; set; }
}

public static void Example()
{
    List<Market> markets = new List<Market>
    {
        new Market { Name = "Emily's", Items = new string[] { "kiwi", "cheery", "banana" } },
        new Market { Name = "Kim's", Items = new string[] { "melon", "mango", "olive" } },
        new Market { Name = "Adam's", Items = new string[] { "kiwi", "apple", "orange" } },
    };

    // Determine which market contains fruit names equal 'kiwi'
    IEnumerable<string> names = from market in markets
                                where market.Items.Contains("kiwi")
                                select market.Name;

    foreach (string name in names)
    {
        Console.WriteLine($"{name} market");
    }

    // This code produces the following output:
    //
    // Emily's market
    // Adam's market
}
```

See also

- [System.Linq](#)
- [Standard Query Operators Overview \(C#\)](#)
- [Dynamically specify predicate filters at run time](#)
- [How to query for sentences that contain a specified set of words \(LINQ\) \(C#\)](#)

Projection operations (C#)

12/28/2021 • 5 minutes to read • [Edit Online](#)

Projection refers to the operation of transforming an object into a new form that often consists only of those properties that will be subsequently used. By using projection, you can construct a new type that is built from each object. You can project a property and perform a mathematical function on it. You can also project the original object without changing it.

The standard query operator methods that perform projection are listed in the following section.

Methods

METHOD NAMES	DESCRIPTION	C# QUERY EXPRESSION SYNTAX	MORE INFORMATION
Select	Projects values that are based on a transform function.	<code>select</code>	Enumerable.Select Queryable.Select
SelectMany	Projects sequences of values that are based on a transform function and then flattens them into one sequence.	Use multiple <code>from</code> clauses	Enumerable.SelectMany Queryable.SelectMany
Zip	Produces a sequence of tuples with elements from 2-3 specified sequences.	Not applicable.	Enumerable.Zip Queryable.Zip

Select

The following example uses the `select` clause to project the first letter from each string in a list of strings.

```
List<string> words = new() { "an", "apple", "a", "day" };

var query = from word in words
            select word.Substring(0, 1);

foreach (string s in query)
    Console.WriteLine(s);

/* This code produces the following output:

    a
    a
    a
    d
*/
```

SelectMany

The following example uses multiple `from` clauses to project each word from each string in a list of strings.

```

List<string> phrases = new() { "an apple a day", "the quick brown fox" };

var query = from phrase in phrases
            from word in phrase.Split(' ')
            select word;

foreach (string s in query)
    Console.WriteLine(s);

/* This code produces the following output:

    an
    apple
    a
    day
    the
    quick
    brown
    fox
*/

```

Zip

There are several overloads for the `Zip` projection operator. All of the `Zip` methods work on sequences of two or more possibly heterogenous types. The first two overloads return tuples, with the corresponding positional type from the given sequences.

Consider the following collections:

```

// An int array with 7 elements.
IEnumerable<int> numbers = new[]
{
    1, 2, 3, 4, 5, 6, 7
};
// A char array with 6 elements.
IEnumerable<char> letters = new[]
{
    'A', 'B', 'C', 'D', 'E', 'F'
};

```

To project these sequences together, use the `Enumerable.Zip<TFirst,TSecond>(IEnumerable<TFirst>, IEnumerable<TSecond>)` operator:

```

foreach ((int number, char letter) in numbers.Zip(letters))
{
    Console.WriteLine($"Number: {number} zipped with letter: '{letter}'");
}
// This code produces the following output:
//     Number: 1 zipped with letter: 'A'
//     Number: 2 zipped with letter: 'B'
//     Number: 3 zipped with letter: 'C'
//     Number: 4 zipped with letter: 'D'
//     Number: 5 zipped with letter: 'E'
//     Number: 6 zipped with letter: 'F'

```

IMPORTANT

The resulting sequence from a zip operation is never longer in length than the shortest sequence. The `numbers` and `letters` collections differ in length, and the resulting sequence omits the last element from the `numbers` collection, as it has nothing to zip with.

The second overload accepts a `third` sequence. Let's create another collection, namely `emoji`:

```
// A string array with 8 elements.
IEnumerable<string> emoji = new[]
{
    "🐼", "🐼", "🐼", "🐼", "🐼", "🐼", "✔️", "🐼"
};
```

To project these sequences together, use the `Enumerable.Zip<TFirst,TSecond,TThird>(IEnumerable<TFirst>, IEnumerable<TSecond>, IEnumerable<TThird>)` operator:

```
foreach ((int number, char letter, string em) in numbers.Zip(letters, emoji))
{
    Console.WriteLine(
        $"Number: {number} is zipped with letter: '{letter}' and emoji: {em}");
}
// This code produces the following output:
//   Number: 1 is zipped with letter: 'A' and emoji: 🐼
//   Number: 2 is zipped with letter: 'B' and emoji: 🐼
//   Number: 3 is zipped with letter: 'C' and emoji: 🐼
//   Number: 4 is zipped with letter: 'D' and emoji: 🐼
//   Number: 5 is zipped with letter: 'E' and emoji: 🐼
//   Number: 6 is zipped with letter: 'F' and emoji: 🐼
```

Much like the previous overload, the `Zip` method projects a tuple, but this time with three elements.

The third overload accepts a `Func<TFirst, TSecond, TResult>` argument that acts as a results selector. Given the two types from the sequences being zipped, you can project a new resulting sequence.

```
foreach (string result in
    numbers.Zip(letters, (number, letter) => $" {number} = {letter} ({(int)letter})"))
{
    Console.WriteLine(result);
}
// This code produces the following output:
//   1 = A (65)
//   2 = B (66)
//   3 = C (67)
//   4 = D (68)
//   5 = E (69)
//   6 = F (70)
```

With the preceding `Zip` overload, the specified function is applied to the corresponding elements `numbers` and `letter`, producing a sequence of the `string` results.

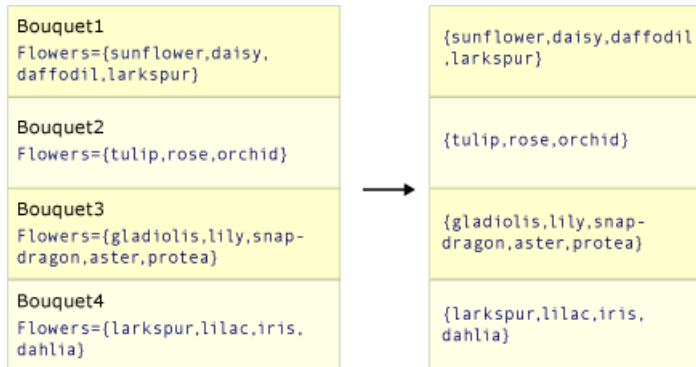
Select VERSUS SelectMany

The work of both `Select` and `SelectMany` is to produce a result value (or values) from source values. `Select` produces one result value for every source value. The overall result is therefore a collection that has the same number of elements as the source collection. In contrast, `SelectMany` produces a single overall result that contains concatenated sub-collections from each source value. The transform function that is passed as an

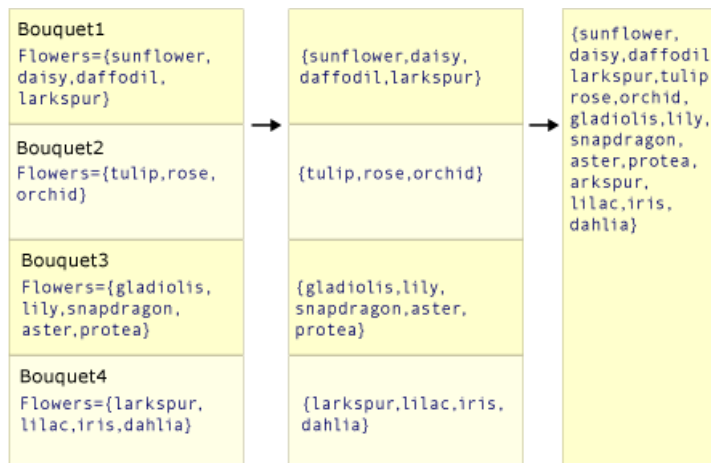
argument to `SelectMany` must return an enumerable sequence of values for each source value. These enumerable sequences are then concatenated by `SelectMany` to create one large sequence.

The following two illustrations show the conceptual difference between the actions of these two methods. In each case, assume that the selector (transform) function selects the array of flowers from each source value.

This illustration depicts how `Select` returns a collection that has the same number of elements as the source collection.



This illustration depicts how `SelectMany` concatenates the intermediate sequence of arrays into one final result value that contains each value from each intermediate array.



Code example

The following example compares the behavior of `Select` and `SelectMany`. The code creates a "bouquet" of flowers by taking the first two items from each list of flower names in the source collection. In this example, the "single value" that the transform function `Select<TSource,TResult>(IEnumerable<TSource>, Func<TSource,TResult>)` uses is itself a collection of values. This requires the extra `foreach` loop in order to enumerate each string in each sub-sequence.

```
class Bouquet
{
    public List<string> Flowers { get; set; }
}

static void SelectVsSelectMany()
{
    List<Bouquet> bouquets = new()
    {
        new Bouquet { Flowers = new List<string> { "sunflower", "daisy", "daffodil", "larkspur" }},
        new Bouquet { Flowers = new List<string> { "tulip", "rose", "orchid" }},
        new Bouquet { Flowers = new List<string> { "gladiolis", "lily", "snapdragon", "aster", "protea" }},
        new Bouquet { Flowers = new List<string> { "larkspur", "lilac", "iris", "dahlia" }}
    };

    IEnumerable<List<string>> query1 = bouquets.Select(bq => bq.Flowers);
```

```

IEnumerable<string> query2 = bouquets.SelectMany(bq => bq.Flowers);

Console.WriteLine("Results by using Select():");
// Note the extra foreach loop here.
foreach (IEnumerable<String> collection in query1)
    foreach (string item in collection)
        Console.WriteLine(item);

Console.WriteLine("\nResults by using SelectMany():");
foreach (string item in query2)
    Console.WriteLine(item);

/* This code produces the following output:

Results by using Select():
sunflower
daisy
daffodil
larkspur
tulip
rose
orchid
gladiolis
lily
snapdragon
aster
protea
larkspur
lilac
iris
dahlia

Results by using SelectMany():
sunflower
daisy
daffodil
larkspur
tulip
rose
orchid
gladiolis
lily
snapdragon
aster
protea
larkspur
lilac
iris
dahlia

*/
}

```

See also

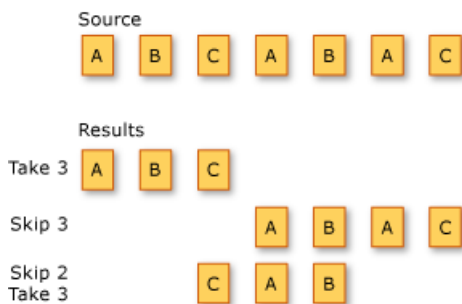
- [System.Linq](#)
- [Standard Query Operators Overview \(C#\)](#)
- [select clause](#)
- [How to populate object collections from multiple sources \(LINQ\) \(C#\)](#)
- [How to split a file into many files by using groups \(LINQ\) \(C#\)](#)

Partitioning data (C#)

12/28/2021 • 2 minutes to read • [Edit Online](#)

Partitioning in LINQ refers to the operation of dividing an input sequence into two sections, without rearranging the elements, and then returning one of the sections.

The following illustration shows the results of three different partitioning operations on a sequence of characters. The first operation returns the first three elements in the sequence. The second operation skips the first three elements and returns the remaining elements. The third operation skips the first two elements in the sequence and returns the next three elements.



The standard query operator methods that partition sequences are listed in the following section.

Operators

METHOD NAMES	DESCRIPTION	C# QUERY EXPRESSION SYNTAX	MORE INFORMATION
Skip	Skips elements up to a specified position in a sequence.	Not applicable.	Enumerable.Skip Queryable.Skip
SkipWhile	Skips elements based on a predicate function until an element does not satisfy the condition.	Not applicable.	Enumerable.SkipWhile Queryable.SkipWhile
Take	Takes elements up to a specified position in a sequence.	Not applicable.	Enumerable.Take Queryable.Take
TakeWhile	Takes elements based on a predicate function until an element does not satisfy the condition.	Not applicable.	Enumerable.TakeWhile Queryable.TakeWhile
Chunk	Splits the elements of a sequence into chunks of a specified maximum size.	Not applicable.	Enumerable.Chunk Queryable.Chunk

Example

The `chunk` operator is used to split elements of a sequence based on a given `size`.

```

int chunkNumber = 1;
foreach (int[] chunk in Enumerable.Range(0, 8).Chunk(3))
{
    Console.WriteLine($"Chunk {chunkNumber++}:");
    foreach (int item in chunk)
    {
        Console.WriteLine($"    {item}");
    }

    Console.WriteLine();
}
// This code produces the following output:
// Chunk 1:
//     0
//     1
//     2
//
//Chunk 2:
//     3
//     4
//     5
//
//Chunk 3:
//     6
//     7

```

The preceding C# code:

- Relies on [Enumerable.Range\(Int32, Int32\)](#) to generate a sequence of numbers.
- Applies the `Chunk` operator, splitting the sequence into chunks with a max size of three.

See also

- [System.Linq](#)
- [Standard Query Operators Overview \(C#\)](#)

Join Operations (C#)

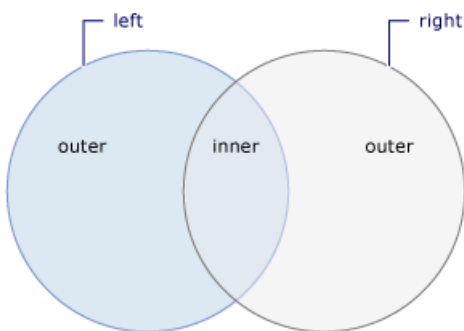
12/28/2021 • 4 minutes to read • [Edit Online](#)

A *join* of two data sources is the association of objects in one data source with objects that share a common attribute in another data source.

Joining is an important operation in queries that target data sources whose relationships to each other cannot be followed directly. In object-oriented programming, this could mean a correlation between objects that is not modeled, such as the backwards direction of a one-way relationship. An example of a one-way relationship is a Customer class that has a property of type City, but the City class does not have a property that is a collection of Customer objects. If you have a list of City objects and you want to find all the customers in each city, you could use a join operation to find them.

The join methods provided in the LINQ framework are [Join](#) and [GroupJoin](#). These methods perform equijoins, or joins that match two data sources based on equality of their keys. (For comparison, Transact-SQL supports join operators other than 'equals', for example the 'less than' operator.) In relational database terms, [Join](#) implements an inner join, a type of join in which only those objects that have a match in the other data set are returned. The [GroupJoin](#) method has no direct equivalent in relational database terms, but it implements a superset of inner joins and left outer joins. A left outer join is a join that returns each element of the first (left) data source, even if it has no correlated elements in the other data source.

The following illustration shows a conceptual view of two sets and the elements within those sets that are included in either an inner join or a left outer join.



Methods

METHOD NAME	DESCRIPTION	C# QUERY EXPRESSION SYNTAX	MORE INFORMATION
Join	Joins two sequences based on key selector functions and extracts pairs of values.	<pre>join ... in ... on ... equals ...</pre>	Enumerable.Join Queryable.Join
GroupJoin	Joins two sequences based on key selector functions and groups the resulting matches for each element.	<pre>join ... in ... on ... equals ... into ...</pre>	Enumerable.GroupJoin Queryable.GroupJoin

Query expression syntax examples

Join

The following example uses the `join ... in ... on ... equals ...` clause to join two sequences based on specific value:

```

class Product
{
    public string Name { get; set; }
    public int CategoryId { get; set; }
}

class Category
{
    public int Id { get; set; }
    public string CategoryName { get; set; }
}

public static void Example()
{
    List<Product> products = new List<Product>
    {
        new Product { Name = "Cola", CategoryId = 0 },
        new Product { Name = "Tea", CategoryId = 0 },
        new Product { Name = "Apple", CategoryId = 1 },
        new Product { Name = "Kiwi", CategoryId = 1 },
        new Product { Name = "Carrot", CategoryId = 2 },
    };

    List<Category> categories = new List<Category>
    {
        new Category { Id = 0, CategoryName = "Beverage" },
        new Category { Id = 1, CategoryName = "Fruit" },
        new Category { Id = 2, CategoryName = "Vegetable" }
    };

    // Join products and categories based on CategoryId
    var query = from product in products
                join category in categories on product.CategoryId equals category.Id
                select new { product.Name, category.CategoryName };

    foreach (var item in query)
    {
        Console.WriteLine($"{item.Name} - {item.CategoryName}");
    }

    // This code produces the following output:
    //
    // Cola - Beverage
    // Tea - Beverage
    // Apple - Fruit
    // Kiwi - Fruit
    // Carrot - Vegetable
}

```

GroupJoin

The following example uses the `join ... in ... on ... equals ... into ...` clause to join two sequences based on specific value and groups the resulting matches for each element:

```

class Product
{
    public string Name { get; set; }
    public int CategoryId { get; set; }
}

class Category
{
    public int Id { get; set; }
    public string CategoryName { get; set; }
}

public static void Example()
{
    List<Product> products = new List<Product>
    {
        new Product { Name = "Cola", CategoryId = 0 },
        new Product { Name = "Tea", CategoryId = 0 },
        new Product { Name = "Apple", CategoryId = 1 },
        new Product { Name = "Kiwi", CategoryId = 1 },
        new Product { Name = "Carrot", CategoryId = 2 },
    };

    List<Category> categories = new List<Category>
    {
        new Category { Id = 0, CategoryName = "Beverage" },
        new Category { Id = 1, CategoryName = "Fruit" },
        new Category { Id = 2, CategoryName = "Vegetable" }
    };

    // Join categories and product based on CategoryId and grouping result
    var productGroups = from category in categories
                        join product in products on category.Id equals product.CategoryId into productGroup
                        select productGroup;

    foreach (IEnumerable<Product> productGroup in productGroups)
    {
        Console.WriteLine("Group");
        foreach (Product product in productGroup)
        {
            Console.WriteLine($"{product.Name,8}");
        }
    }

    // This code produces the following output:
    //
    // Group
    //   Cola
    //   Tea
    // Group
    //   Apple
    //   Kiwi
    // Group
    //   Carrot
}

```

See also

- [System.Linq](#)
- [Standard Query Operators Overview \(C#\)](#)
- [Anonymous Types](#)
- [Formulate Joins and Cross-Product Queries](#)
- [join clause](#)

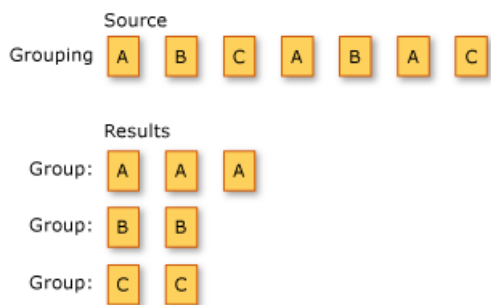
- [Join by using composite keys](#)
- [How to join content from dissimilar files \(LINQ\) \(C#\)](#)
- [Order the results of a join clause](#)
- [Perform custom join operations](#)
- [Perform grouped joins](#)
- [Perform inner joins](#)
- [Perform left outer joins](#)
- [How to populate object collections from multiple sources \(LINQ\) \(C#\)](#)

Grouping Data (C#)

12/28/2021 • 2 minutes to read • [Edit Online](#)

Grouping refers to the operation of putting data into groups so that the elements in each group share a common attribute.

The following illustration shows the results of grouping a sequence of characters. The key for each group is the character.



The standard query operator methods that group data elements are listed in the following section.

Methods

METHOD NAME	DESCRIPTION	C# QUERY EXPRESSION SYNTAX	MORE INFORMATION
GroupBy	Groups elements that share a common attribute. Each group is represented by an IGrouping<TKey,TElement> object.	<code>group ... by</code> -or- <code>group ... by ... into ...</code>	Enumerable.GroupBy Queryable.GroupBy
ToLookup	Inserts elements into a Lookup<TKey,TElement> (a one-to-many dictionary) based on a key selector function.	Not applicable.	Enumerable.ToLookup

Query Expression Syntax Example

The following code example uses the `group by` clause to group integers in a list according to whether they are even or odd.

```

List<int> numbers = new List<int>() { 35, 44, 200, 84, 3987, 4, 199, 329, 446, 208 };

IEnumerable<IGrouping<int, int>> query = from number in numbers
                                         group number by number % 2;

foreach (var group in query)
{
    Console.WriteLine(group.Key == 0 ? "\nEven numbers:" : "\nOdd numbers:");
    foreach (int i in group)
        Console.WriteLine(i);
}

/* This code produces the following output:

    Odd numbers:
    35
    3987
    199
    329

    Even numbers:
    44
    200
    84
    4
    446
    208

*/

```

See also

- [System.Linq](#)
- [Standard Query Operators Overview \(C#\)](#)
- [group clause](#)
- [Create a nested group](#)
- [How to group files by extension \(LINQ\) \(C#\)](#)
- [Group query results](#)
- [Perform a subquery on a grouping operation](#)
- [How to split a file into many files by using groups \(LINQ\) \(C#\)](#)

Generation Operations (C#)

12/28/2021 • 2 minutes to read • [Edit Online](#)

Generation refers to creating a new sequence of values.

The standard query operator methods that perform generation are listed in the following section.

Methods

METHOD NAME	DESCRIPTION	C# QUERY EXPRESSION SYNTAX	MORE INFORMATION
DefaultIfEmpty	Replaces an empty collection with a default valued singleton collection.	Not applicable.	Enumerable.DefaultIfEmpty Queryable.DefaultIfEmpty
Empty	Returns an empty collection.	Not applicable.	Enumerable.Empty
Range	Generates a collection that contains a sequence of numbers.	Not applicable.	Enumerable.Range
Repeat	Generates a collection that contains one repeated value.	Not applicable.	Enumerable.Repeat

See also

- [System.Linq](#)
- [Standard Query Operators Overview \(C#\)](#)

Equality Operations (C#)

12/28/2021 • 2 minutes to read • [Edit Online](#)

Two sequences whose corresponding elements are equal and which have the same number of elements are considered equal.

Methods

METHOD NAME	DESCRIPTION	C# QUERY EXPRESSION SYNTAX	MORE INFORMATION
SequenceEqual	Determines whether two sequences are equal by comparing elements in a pair-wise manner.	Not applicable.	Enumerable.SequenceEqual Queryable.SequenceEqual

See also

- [System.Linq](#)
- [Standard Query Operators Overview \(C#\)](#)
- [How to compare the contents of two folders \(LINQ\) \(C#\)](#)

Element operations (C#)

12/28/2021 • 2 minutes to read • [Edit Online](#)

Element operations return a single, specific element from a sequence.

The standard query operator methods that perform element operations are listed in the following section.

Methods

METHOD NAME	DESCRIPTION	C# QUERY EXPRESSION SYNTAX	MORE INFORMATION
ElementAt	Returns the element at a specified index in a collection.	Not applicable.	Enumerable.ElementAt Queryable.ElementAt
ElementAtOrDefault	Returns the element at a specified index in a collection or a default value if the index is out of range.	Not applicable.	Enumerable.ElementAtOrDefault Queryable.ElementAtOrDefault
First	Returns the first element of a collection, or the first element that satisfies a condition.	Not applicable.	Enumerable.First Queryable.First
FirstOrDefault	Returns the first element of a collection, or the first element that satisfies a condition. Returns a default value if no such element exists.	Not applicable.	Enumerable.FirstOrDefault Queryable.FirstOrDefault Queryable.FirstOrDefault<TSource> (IQueryable<TSource>)
Last	Returns the last element of a collection, or the last element that satisfies a condition.	Not applicable.	Enumerable.Last Queryable.Last
LastOrDefault	Returns the last element of a collection, or the last element that satisfies a condition. Returns a default value if no such element exists.	Not applicable.	Enumerable.LastOrDefault Queryable.LastOrDefault
Single	Returns the only element of a collection or the only element that satisfies a condition. Throws an InvalidOperationException if there is no element or more than one element to return.	Not applicable.	Enumerable.Single Queryable.Single

METHOD NAME	DESCRIPTION	C# QUERY EXPRESSION SYNTAX	MORE INFORMATION
SingleOrDefault	Returns the only element of a collection or the only element that satisfies a condition. Returns a default value if there is no element to return. Throws an InvalidOperationException if there is more than one element to return.	Not applicable.	Enumerable.SingleOrDefault Queryable.SingleOrDefault

See also

- [System.Linq](#)
- [Standard Query Operators Overview \(C#\)](#)
- [How to query for the largest file or files in a directory tree \(LINQ\) \(C#\)](#)

Converting Data Types (C#)

12/28/2021 • 2 minutes to read • [Edit Online](#)

Conversion methods change the type of input objects.

Conversion operations in LINQ queries are useful in a variety of applications. Following are some examples:

- The [Enumerable.AsEnumerable](#) method can be used to hide a type's custom implementation of a standard query operator.
- The [Enumerable.OfType](#) method can be used to enable non-parameterized collections for LINQ querying.
- The [Enumerable.ToArray](#), [Enumerable.ToDictionary](#), [Enumerable.ToList](#), and [Enumerable.ToLookup](#) methods can be used to force immediate query execution instead of deferring it until the query is enumerated.

Methods

The following table lists the standard query operator methods that perform data-type conversions.

The conversion methods in this table whose names start with "As" change the static type of the source collection but do not enumerate it. The methods whose names start with "To" enumerate the source collection and put the items into the corresponding collection type.

METHOD NAME	DESCRIPTION	C# QUERY EXPRESSION SYNTAX	MORE INFORMATION
AsEnumerable	Returns the input typed as IEnumerable<T> .	Not applicable.	Enumerable.AsEnumerable
AsQueryable	Converts a (generic) IEnumerable to a (generic) IQueryable .	Not applicable.	Queryable.AsQueryable
Cast	Casts the elements of a collection to a specified type.	Use an explicitly typed range variable. For example: <div>from string str in words</div>	Enumerable.Cast Queryable.Cast
OfType	Filters values, depending on their ability to be cast to a specified type.	Not applicable.	Enumerable.OfType Queryable.OfType
ToArray	Converts a collection to an array. This method forces query execution.	Not applicable.	Enumerable.ToArray
ToDictionary	Puts elements into a Dictionary<TKey,TValue> based on a key selector function. This method forces query execution.	Not applicable.	Enumerable.ToDictionary

METHOD NAME	DESCRIPTION	C# QUERY EXPRESSION SYNTAX	MORE INFORMATION
ToList	Converts a collection to a List<T> . This method forces query execution.	Not applicable.	Enumerable.ToList
ToLookup	Puts elements into a Lookup<TKey,TElement> (a one-to-many dictionary) based on a key selector function. This method forces query execution.	Not applicable.	Enumerable.ToLookup

Query Expression Syntax Example

The following code example uses an explicitly typed range variable to cast a type to a subtype before accessing a member that is available only on the subtype.

```
class Plant
{
    public string Name { get; set; }
}

class CarnivorousPlant : Plant
{
    public string TrapType { get; set; }
}

static void Cast()
{
    Plant[] plants = new Plant[] {
        new CarnivorousPlant { Name = "Venus Fly Trap", TrapType = "Snap Trap" },
        new CarnivorousPlant { Name = "Pitcher Plant", TrapType = "Pitfall Trap" },
        new CarnivorousPlant { Name = "Sundew", TrapType = "Flypaper Trap" },
        new CarnivorousPlant { Name = "Waterwheel Plant", TrapType = "Snap Trap" }
    };

    var query = from CarnivorousPlant cPlant in plants
                where cPlant.TrapType == "Snap Trap"
                select cPlant;

    foreach (Plant plant in query)
        Console.WriteLine(plant.Name);

    /* This code produces the following output:

        Venus Fly Trap
        Waterwheel Plant
    */
}
```

See also

- [System.Linq](#)
- [Standard Query Operators Overview \(C#\)](#)
- [from clause](#)
- [LINQ Query Expressions](#)
- [How to query an ArrayList with LINQ \(C#\)](#)

Concatenation Operations (C#)

12/28/2021 • 2 minutes to read • [Edit Online](#)

Concatenation refers to the operation of appending one sequence to another.

The following illustration depicts a concatenation operation on two sequences of characters.



The standard query operator methods that perform concatenation are listed in the following section.

Methods

METHOD NAME	DESCRIPTION	C# QUERY EXPRESSION SYNTAX	MORE INFORMATION
Concat	Concatenates two sequences to form one sequence.	Not applicable.	Enumerable.Concat Queryable.Concat

See also

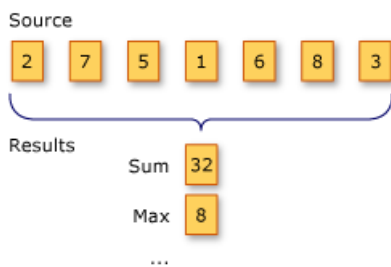
- [System.Linq](#)
- [Standard Query Operators Overview \(C#\)](#)
- [How to combine and compare string collections \(LINQ\) \(C#\)](#)

Aggregation operations (C#)

12/28/2021 • 2 minutes to read • [Edit Online](#)

An aggregation operation computes a single value from a collection of values. An example of an aggregation operation is calculating the average daily temperature from a month's worth of daily temperature values.

The following illustration shows the results of two different aggregation operations on a sequence of numbers. The first operation sums the numbers. The second operation returns the maximum value in the sequence.



The standard query operator methods that perform aggregation operations are listed in the following section.

Methods

METHOD NAME	DESCRIPTION	C# QUERY EXPRESSION SYNTAX	MORE INFORMATION
Aggregate	Performs a custom aggregation operation on the values of a collection.	Not applicable.	Enumerable.Aggregate Queryable.Aggregate
Average	Calculates the average value of a collection of values.	Not applicable.	Enumerable.Average Queryable.Average
Count	Counts the elements in a collection, optionally only those elements that satisfy a predicate function.	Not applicable.	Enumerable.Count Queryable.Count
LongCount	Counts the elements in a large collection, optionally only those elements that satisfy a predicate function.	Not applicable.	Enumerable.LongCount Queryable.LongCount
Max or MaxBy	Determines the maximum value in a collection.	Not applicable.	Enumerable.Max Enumerable.MaxBy Queryable.Max Queryable.MaxBy
Min or MinBy	Determines the minimum value in a collection.	Not applicable.	Enumerable.Min Enumerable.MinBy Queryable.Min Queryable.MinBy

METHOD NAME	DESCRIPTION	C# QUERY EXPRESSION SYNTAX	MORE INFORMATION
Sum	Calculates the sum of the values in a collection.	Not applicable.	Enumerable.Sum Queryable.Sum

See also

- [System.Linq](#)
- [Standard Query Operators Overview \(C#\)](#)
- [How to compute column values in a CSV text file \(LINQ\) \(C#\)](#)
- [How to query for the largest file or files in a directory tree \(LINQ\) \(C#\)](#)
- [How to query for the total number of bytes in a set of folders \(LINQ\) \(C#\)](#)

LINQ to Objects (C#)

12/28/2021 • 2 minutes to read • [Edit Online](#)

The term "LINQ to Objects" refers to the use of LINQ queries with any [IEnumerable](#) or [IEnumerable<T>](#) collection directly, without the use of an intermediate LINQ provider or API such as [LINQ to SQL](#) or [LINQ to XML](#). You can use LINQ to query any enumerable collections such as [List<T>](#), [Array](#), or [Dictionary<TKey,TValue>](#). The collection may be user-defined or may be returned by a .NET API.

In a basic sense, LINQ to Objects represents a new approach to collections. In the old way, you had to write complex `foreach` loops that specified how to retrieve data from a collection. In the LINQ approach, you write declarative code that describes what you want to retrieve.

In addition, LINQ queries offer three main advantages over traditional `foreach` loops:

- They are more concise and readable, especially when filtering multiple conditions.
- They provide powerful filtering, ordering, and grouping capabilities with a minimum of application code.
- They can be ported to other data sources with little or no modification.

In general, the more complex the operation you want to perform on the data, the more benefit you'll realize by using LINQ instead of traditional iteration techniques.

The purpose of this section is to demonstrate the LINQ approach with some select examples. It's not intended to be exhaustive.

In This Section

[LINQ and Strings \(C#\)](#)

Explains how LINQ can be used to query and transform strings and collections of strings. Also includes links to articles that demonstrate these principles.

[LINQ and Reflection \(C#\)](#)

Links to a sample that demonstrates how LINQ uses reflection.

[LINQ and File Directories \(C#\)](#)

Explains how LINQ can be used to interact with file systems. Also includes links to articles that demonstrate these concepts.

[How to query an ArrayList with LINQ \(C#\)](#)

Demonstrates how to query an ArrayList in C#.

[How to add custom methods for LINQ queries \(C#\)](#)

Explains how to extend the set of methods that you can use for LINQ queries by adding extension methods to the [IEnumerable<T>](#) interface.

[Language-Integrated Query \(LINQ\) \(C#\)](#)

Provides links to articles that explain LINQ and provide examples of code that perform queries.

LINQ and strings (C#)

12/28/2021 • 2 minutes to read • [Edit Online](#)

LINQ can be used to query and transform strings and collections of strings. It can be especially useful with semi-structured data in text files. LINQ queries can be combined with traditional string functions and regular expressions. For example, you can use the [String.Split](#) or [Regex.Split](#) method to create an array of strings that you can then query or modify by using LINQ. You can use the [Regex.IsMatch](#) method in the `where` clause of a LINQ query. And you can use LINQ to query or modify the [MatchCollection](#) results returned by a regular expression.

You can also use the techniques described in this section to transform semi-structured text data to XML. For more information, see [How to generate XML from CSV files](#).

The examples in this section fall into two categories:

Querying a block of text

You can query, analyze, and modify text blocks by splitting them into a queryable array of smaller strings by using the [String.Split](#) method or the [Regex.Split](#) method. You can split the source text into words, sentences, paragraphs, pages, or any other criteria, and then perform additional splits if they are required in your query.

- [How to count occurrences of a word in a string \(LINQ\) \(C#\)](#)
Shows how to use LINQ for simple querying over text.
- [How to query for sentences that contain a specified set of words \(LINQ\) \(C#\)](#)
Shows how to split text files on arbitrary boundaries and how to perform queries against each part.
- [How to query for characters in a string \(LINQ\) \(C#\)](#)
Demonstrates that a string is a queryable type.
- [How to combine LINQ queries with regular expressions \(C#\)](#)
Shows how to use regular expressions in LINQ queries for complex pattern matching on filtered query results.

Querying semi-structured data in text format

Many different types of text files consist of a series of lines, often with similar formatting, such as tab- or comma-delimited files or fixed-length lines. After you read such a text file into memory, you can use LINQ to query and/or modify the lines. LINQ queries also simplify the task of combining data from multiple sources.

- [How to find the set difference between two lists \(LINQ\) \(C#\)](#)
Shows how to find all the strings that are present in one list but not the other.
- [How to sort or filter text data by any word or field \(LINQ\) \(C#\)](#)
Shows how to sort text lines based on any word or field.
- [How to reorder the fields of a delimited file \(LINQ\) \(C#\)](#)
Shows how to reorder fields in a line in a .csv file.
- [How to combine and compare string collections \(LINQ\) \(C#\)](#)

Shows how to combine string lists in various ways.

- [How to populate object collections from multiple sources \(LINQ\) \(C#\)](#)

Shows how to create object collections by using multiple text files as data sources.

- [How to join content from dissimilar files \(LINQ\) \(C#\)](#)

Shows how to combine strings in two lists into a single string by using a matching key.

- [How to split a file into many files by using groups \(LINQ\) \(C#\)](#)

Shows how to create new files by using a single file as a data source.

- [How to compute column values in a CSV text file \(LINQ\) \(C#\)](#)

Shows how to perform mathematical computations on text data in .csv files.

See also

- [Language-Integrated Query \(LINQ\) \(C#\)](#)
- [How to generate XML from CSV files](#)

How to count occurrences of a word in a string (LINQ) (C#)

12/28/2021 • 2 minutes to read • [Edit Online](#)

This example shows how to use a LINQ query to count the occurrences of a specified word in a string. Note that to perform the count, first the [Split](#) method is called to create an array of words. There is a performance cost to the [Split](#) method. If the only operation on the string is to count the words, you should consider using the [Matches](#) or [IndexOf](#) methods instead. However, if performance is not a critical issue, or you have already split the sentence in order to perform other types of queries over it, then it makes sense to use LINQ to count the words or phrases as well.

Example

```
class CountWords
{
    static void Main()
    {
        string text = @"Historically, the world of data and the world of objects" +
            @" have not been well integrated. Programmers work in C# or Visual Basic" +
            @" and also in SQL or XQuery. On the one side are concepts such as classes," +
            @" objects, fields, inheritance, and .NET APIs. On the other side" +
            @" are tables, columns, rows, nodes, and separate languages for dealing with" +
            @" them. Data types often require translation between the two worlds; there are" +
            @" different standard functions. Because the object world has no notion of query, a" +
            @" query can only be represented as a string without compile-time type checking or" +
            @" IntelliSense support in the IDE. Transferring data from SQL tables or XML trees to" +
            @" objects in memory is often tedious and error-prone.";

        string searchTerm = "data";

        //Convert the string into an array of words
        string[] source = text.Split(new char[] { '.', '?', '!', ' ', ';', ':', ',' },
            StringSplitOptions.RemoveEmptyEntries);

        // Create the query. Use ToLowerInvariant to match "data" and "Data"
        var matchQuery = from word in source
            where word.ToLowerInvariant() == searchTerm.ToLowerInvariant()
            select word;

        // Count the matches, which executes the query.
        int wordCount = matchQuery.Count();
        Console.WriteLine("{0} occurrences(s) of the search term \"{1}\" were found.", wordCount,
            searchTerm);

        // Keep console window open in debug mode
        Console.WriteLine("Press any key to exit");
        Console.ReadKey();
    }
}

/* Output:
3 occurrences(s) of the search term "data" were found.
*/
```

Compiling the Code

Create a C# console application project, with `using` directives for the `System.Linq` and `System.IO` namespaces.

See also

- [LINQ and Strings \(C#\)](#)

How to query for sentences that contain a specified set of words (LINQ) (C#)

12/28/2021 • 2 minutes to read • [Edit Online](#)

This example shows how to find sentences in a text file that contain matches for each of a specified set of words. Although the array of search terms is hard-coded in this example, it could also be populated dynamically at run time. In this example, the query returns the sentences that contain the words "Historically," "data," and "integrated."

Example

```
class FindSentences
{
    static void Main()
    {
        string text = @"Historically, the world of data and the world of objects " +
            @"have not been well integrated. Programmers work in C# or Visual Basic " +
            @"and also in SQL or XQuery. On the one side are concepts such as classes, " +
            @"objects, fields, inheritance, and .NET APIs. On the other side " +
            @"are tables, columns, rows, nodes, and separate languages for dealing with " +
            @"them. Data types often require translation between the two worlds; there are " +
            @"different standard functions. Because the object world has no notion of query, a " +
            @"query can only be represented as a string without compile-time type checking or " +
            @"IntelliSense support in the IDE. Transferring data from SQL tables or XML trees to " +
            @"objects in memory is often tedious and error-prone.";

        // Split the text block into an array of sentences.
        string[] sentences = text.Split(new char[] { '.', '?', '!' });

        // Define the search terms. This list could also be dynamically populated at run time.
        string[] wordsToMatch = { "Historically", "data", "integrated" };

        // Find sentences that contain all the terms in the wordsToMatch array.
        // Note that the number of terms to match is not specified at compile time.
        var sentenceQuery = from sentence in sentences
            let w = sentence.Split(new char[] { '.', '?', '!', ' ', ';', ':', ',' },
                StringSplitOptions.RemoveEmptyEntries)
            where w.Distinct().Intersect(wordsToMatch).Count() == wordsToMatch.Count()
            select sentence;

        // Execute the query. Note that you can explicitly type
        // the iteration variable here even though sentenceQuery
        // was implicitly typed.
        foreach (string str in sentenceQuery)
        {
            Console.WriteLine(str);
        }

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit");
        Console.ReadKey();
    }
}

/* Output:
Historically, the world of data and the world of objects have not been well integrated
*/
```

The query works by first splitting the text into sentences, and then splitting the sentences into an array of strings

that hold each word. For each of these arrays, the [Distinct](#) method removes all duplicate words, and then the query performs an [Intersect](#) operation on the word array and the `wordsToMatch` array. If the count of the intersection is the same as the count of the `wordsToMatch` array, all words were found in the words and the original sentence is returned.

In the call to [Split](#), the punctuation marks are used as separators in order to remove them from the string. If you did not do this, for example you could have a string "Historically," that would not match "Historically" in the `wordsToMatch` array. You may have to use additional separators, depending on the types of punctuation found in the source text.

Compiling the Code

Create a C# console application project, with `using` directives for the System.Linq and System.IO namespaces.

See also

- [LINQ and Strings \(C#\)](#)

How to query for characters in a string (LINQ) (C#)

12/28/2021 • 2 minutes to read • [Edit Online](#)

Because the [String](#) class implements the generic [IEnumerable<T>](#) interface, any string can be queried as a sequence of characters. However, this is not a common use of LINQ. For complex pattern matching operations, use the [Regex](#) class.

Example

The following example queries a string to determine the number of numeric digits it contains. Note that the query is "reused" after it is executed the first time. This is possible because the query itself does not store any actual results.

```
class QueryAString
{
    static void Main()
    {
        string aString = "ABCDE99F-J74-12-89A";

        // Select only those characters that are numbers
        IEnumerable<char> stringQuery =
            from ch in aString
            where Char.IsDigit(ch)
            select ch;

        // Execute the query
        foreach (char c in stringQuery)
            Console.Write(c + " ");

        // Call the Count method on the existing query.
        int count = stringQuery.Count();
        Console.WriteLine("Count = {0}", count);

        // Select all characters before the first '-'
        IEnumerable<char> stringQuery2 = aString.TakeWhile(c => c != '-');

        // Execute the second query
        foreach (char c in stringQuery2)
            Console.Write(c);

        Console.WriteLine(System.Environment.NewLine + "Press any key to exit");
        Console.ReadKey();
    }
}

/* Output:
Output: 9 9 7 4 1 2 8 9
Count = 8
ABCDE99F
*/
```

Compiling the Code

Create a C# console application project, with `using` directives for the `System.Linq` and `System.IO` namespaces.

See also

- [LINQ and Strings \(C#\)](#)
- [How to combine LINQ queries with regular expressions \(C#\)](#)

How to combine LINQ queries with regular expressions (C#)

12/28/2021 • 2 minutes to read • [Edit Online](#)

This example shows how to use the [Regex](#) class to create a regular expression for more complex matching in text strings. The LINQ query makes it easy to filter on exactly the files that you want to search with the regular expression, and to shape the results.

Example

```
class QueryWithRegex
{
    public static void Main()
    {
        // Modify this path as necessary so that it accesses your version of Visual Studio.
        string startFolder = @"C:\Program Files (x86)\Microsoft Visual Studio 14.0\";
        // One of the following paths may be more appropriate on your computer.
        //string startFolder = @"C:\Program Files (x86)\Microsoft Visual Studio\2017\";

        // Take a snapshot of the file system.
        IEnumerable<System.IO.FileInfo> fileList = GetFiles(startFolder);

        // Create the regular expression to find all things "Visual".
        System.Text.RegularExpressions.Regex searchTerm =
            new System.Text.RegularExpressions.Regex(@"Visual (Basic|C#|C\+\+|Studio)");

        // Search the contents of each .htm file.
        // Remove the where clause to find even more matchedValues!
        // This query produces a list of files where a match
        // was found, and a list of the matchedValues in that file.
        // Note: Explicit typing of "Match" in select clause.
        // This is required because MatchCollection is not a
        // generic IEnumerable collection.
        var queryMatchingFiles =
            from file in fileList
            where file.Extension == ".htm"
            let fileText = System.IO.File.ReadAllText(file.FullName)
            let matches = searchTerm.Matches(fileText)
            where matches.Count > 0
            select new
            {
                name = file.FullName,
                matchedValues = from System.Text.RegularExpressions.Match match in matches
                               select match.Value
            };

        // Execute the query.
        Console.WriteLine("The term \"{0}\" was found in:", searchTerm.ToString());

        foreach (var v in queryMatchingFiles)
        {
            // Trim the path a bit, then write
            // the file name in which a match was found.
            string s = v.name.Substring(startFolder.Length - 1);
            Console.WriteLine(s);

            // For this file, write out all the matching strings
            foreach (var v2 in v.matchedValues)
            {
```

```

        Console.WriteLine("  " + v2);
    }
}

// Keep the console window open in debug mode
Console.WriteLine("Press any key to exit");
Console.ReadKey();
}

// This method assumes that the application has discovery
// permissions for all folders under the specified path.
static IEnumerable<System.IO.FileInfo> GetFiles(string path)
{
    if (!System.IO.Directory.Exists(path))
        throw new System.IO.DirectoryNotFoundException();

    string[] fileNames = null;
    List<System.IO.FileInfo> files = new List<System.IO.FileInfo>();

    fileNames = System.IO.Directory.GetFiles(path, " *.*", System.IO.SearchOption.AllDirectories);
    foreach (string name in fileNames)
    {
        files.Add(new System.IO.FileInfo(name));
    }
    return files;
}
}

```

Note that you can also query the [MatchCollection](#) object that is returned by a `Regex` search. In this example only the value of each match is produced in the results. However, it is also possible to use LINQ to perform all kinds of filtering, sorting, and grouping on that collection. Because [MatchCollection](#) is a non-generic [IEnumerable](#) collection, you have to explicitly state the type of the range variable in the query.

Compiling the Code

Create a C# console application project with `using` directives for the `System.Linq` and `System.IO` namespaces.

See also

- [LINQ and Strings \(C#\)](#)
- [LINQ and File Directories \(C#\)](#)

How to find the set difference between two lists (LINQ) (C#)

12/28/2021 • 2 minutes to read • [Edit Online](#)

This example shows how to use LINQ to compare two lists of strings and output those lines that are in names1.txt but not in names2.txt.

To create the data files

1. Copy names1.txt and names2.txt to your solution folder as shown in [How to combine and compare string collections \(LINQ\) \(C#\)](#).

Example

```
class CompareLists
{
    static void Main()
    {
        // Create the IEnumerable data sources.
        string[] names1 = System.IO.File.ReadAllLines(@"../../names1.txt");
        string[] names2 = System.IO.File.ReadAllLines(@"../../names2.txt");

        // Create the query. Note that method syntax must be used here.
        IEnumerable<string> differenceQuery =
            names1.Except(names2);

        // Execute the query.
        Console.WriteLine("The following lines are in names1.txt but not names2.txt");
        foreach (string s in differenceQuery)
            Console.WriteLine(s);

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit");
        Console.ReadKey();
    }
}
/* Output:
    The following lines are in names1.txt but not names2.txt
    Potra, Cristina
    Noriega, Fabricio
    Aw, Kam Foo
    Toyoshima, Tim
    Guy, Wey Yuan
    Garcia, Debra
*/
```

Some types of query operations in C#, such as [Except](#), [Distinct](#), [Union](#), and [Concat](#), can only be expressed in method-based syntax.

Compiling the Code

Create a C# console application project, with `using` directives for the System.Linq and System.IO namespaces.

See also

- [LINQ and Strings \(C#\)](#)

How to sort or filter text data by any word or field (LINQ) (C#)

12/28/2021 • 2 minutes to read • [Edit Online](#)

The following example shows how to sort lines of structured text, such as comma-separated values, by any field in the line. The field may be dynamically specified at run time. Assume that the fields in scores.csv represent a student's ID number, followed by a series of four test scores.

To create a file that contains data

1. Copy the scores.csv data from the topic [How to join content from dissimilar files \(LINQ\) \(C#\)](#) and save it to your solution folder.

Example

```

public class SortLines
{
    static void Main()
    {
        // Create an IEnumerable data source
        string[] scores = System.IO.File.ReadAllLines(@"../.././scores.csv");

        // Change this to any value from 0 to 4.
        int sortField = 1;

        Console.WriteLine("Sorted highest to lowest by field [{0}]:", sortField);

        // Demonstrates how to return query from a method.
        // The query is executed here.
        foreach (string str in RunQuery(scores, sortField))
        {
            Console.WriteLine(str);
        }

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit");
        Console.ReadKey();
    }

    // Returns the query variable, not query results!
    static IEnumerable<string> RunQuery(IEnumerable<string> source, int num)
    {
        // Split the string and sort on field[num]
        var scoreQuery = from line in source
                        let fields = line.Split(',')
                        orderby fields[num] descending
                        select line;

        return scoreQuery;
    }
}

/* Output (if sortField == 1):
Sorted highest to lowest by field [1]:
116, 99, 86, 90, 94
120, 99, 82, 81, 79
111, 97, 92, 81, 60
114, 97, 89, 85, 82
121, 96, 85, 91, 60
122, 94, 92, 91, 91
117, 93, 92, 80, 87
118, 92, 90, 83, 78
113, 88, 94, 65, 91
112, 75, 84, 91, 39
119, 68, 79, 88, 92
115, 35, 72, 91, 70
*/

```

This example also demonstrates how to return a query variable from a method.

Compiling the Code

Create a C# console application project, with `using` directives for the System.Linq and System.IO namespaces.

See also

- [LINQ and Strings \(C#\)](#)

How to reorder the fields of a delimited file (LINQ) (C#)

12/28/2021 • 2 minutes to read • [Edit Online](#)

A comma-separated value (CSV) file is a text file that is often used to store spreadsheet data or other tabular data that is represented by rows and columns. By using the [Split](#) method to separate the fields, it is very easy to query and manipulate CSV files by using LINQ. In fact, the same technique can be used to reorder the parts of any structured line of text; it is not limited to CSV files.

In the following example, assume that the three columns represent students' "last name," "first name", and "ID." The fields are in alphabetical order based on the students' last names. The query produces a new sequence in which the ID column appears first, followed by a second column that combines the student's first name and last name. The lines are reordered according to the ID field. The results are saved into a new file and the original data is not modified.

To create the data file

1. Copy the following lines into a plain text file that is named `spreadsheet1.csv`. Save the file in your project folder.

```
Adams, Terry, 120
Fakhouri, Fadi, 116
Feng, Hanying, 117
Garcia, Cesar, 114
Garcia, Debra, 115
Garcia, Hugo, 118
Mortensen, Sven, 113
O'Donnell, Claire, 112
Omelchenko, Svetlana, 111
Tucker, Lance, 119
Tucker, Michael, 122
Zabokritski, Eugene, 121
```

Example


```

class CSVFiles
{
    static void Main(string[] args)
    {
        // Create the IEnumerable data source
        string[] lines = System.IO.File.ReadAllLines(@"../../../spreadsheet1.csv");

        // Create the query. Put field 2 first, then
        // reverse and combine fields 0 and 1 from the old field
        IEnumerable<string> query =
            from line in lines
            let x = line.Split(',')
            orderby x[2]
            select x[2] + ", " + (x[1] + " " + x[0]);

        // Execute the query and write out the new file. Note that WriteAllLines
        // takes a string[], so ToArray is called on the query.
        System.IO.File.WriteAllLines(@"../../../spreadsheet2.csv", query.ToArray());

        Console.WriteLine("Spreadsheet2.csv written to disk. Press any key to exit");
        Console.ReadKey();
    }
}

/* Output to spreadsheet2.csv:
111, Svetlana Omelchenko
112, Claire O'Donnell
113, Sven Mortensen
114, Cesar Garcia
115, Debra Garcia
116, Fadi Fakhouri
117, Hanying Feng
118, Hugo Garcia
119, Lance Tucker
120, Terry Adams
121, Eugene Zabokritski
122, Michael Tucker
*/

```

Compiling the Code

Create a C# console application project, with `using` directives for the `System.Linq` and `System.IO` namespaces.

See also

- [LINQ and Strings \(C#\)](#)
- [LINQ and File Directories \(C#\)](#)
- [How to generate XML from CSV files \(C#\)](#)

How to combine and compare string collections (LINQ) (C#)

12/28/2021 • 2 minutes to read • [Edit Online](#)

This example shows how to merge files that contain lines of text and then sort the results. Specifically, it shows how to perform a simple concatenation, a union, and an intersection on the two sets of text lines.

To set up the project and the text files

1. Copy these names into a text file that is named names1.txt and save it in your project folder:

```
Bankov, Peter
Holm, Michael
Garcia, Hugo
Potra, Cristina
Noriega, Fabricio
Aw, Kam Foo
Beebe, Ann
Toyoshima, Tim
Guy, Wey Yuan
Garcia, Debra
```

2. Copy these names into a text file that is named names2.txt and save it in your project folder. Note that the two files have some names in common.

```
Liu, Jinghao
Bankov, Peter
Holm, Michael
Garcia, Hugo
Beebe, Ann
Gilchrist, Beth
Myrcha, Jacek
Giakoumakis, Leo
McLin, Nkenge
El Yassir, Mehdi
```

Example

```
class MergeStrings
{
    static void Main(string[] args)
    {
        //Put text files in your solution folder
        string[] fileA = System.IO.File.ReadAllLines(@"../../names1.txt");
        string[] fileB = System.IO.File.ReadAllLines(@"../../names2.txt");

        //Simple concatenation and sort. Duplicates are preserved.
        IEnumerable<string> concatQuery =
            fileA.Concat(fileB).OrderBy(s => s);

        // Pass the query variable to another function for execution.
        OutputQueryResults(concatQuery, "Simple concatenate and sort. Duplicates are preserved.");

        // Concatenate and remove duplicate names based on
        // default string comparer.
        IEnumerable<string> uniqueNamesQuery =
```

```

        fileA.Union(fileB).OrderBy(s => s);
    OutputQueryResults(uniqueNamesQuery, "Union removes duplicate names:");

    // Find the names that occur in both files (based on
    // default string comparer).
    IEnumerable<string> commonNamesQuery =
        fileA.Intersect(fileB);
    OutputQueryResults(commonNamesQuery, "Merge based on intersect:");

    // Find the matching fields in each list. Merge the two
    // results by using Concat, and then
    // sort using the default string comparer.
    string nameMatch = "Garcia";

    IEnumerable<String> tempQuery1 =
        from name in fileA
        let n = name.Split(',')
        where n[0] == nameMatch
        select name;

    IEnumerable<string> tempQuery2 =
        from name2 in fileB
        let n2 = name2.Split(',')
        where n2[0] == nameMatch
        select name2;

    IEnumerable<string> nameMatchQuery =
        tempQuery1.Concat(tempQuery2).OrderBy(s => s);
    OutputQueryResults(nameMatchQuery, $"Concat based on partial name match \"{nameMatch}\":");

    // Keep the console window open in debug mode.
    Console.WriteLine("Press any key to exit");
    Console.ReadKey();
}

static void OutputQueryResults(IEnumerable<string> query, string message)
{
    Console.WriteLine(System.Environment.NewLine + message);
    foreach (string item in query)
    {
        Console.WriteLine(item);
    }
    Console.WriteLine("{0} total names in list", query.Count());
}
}
/* Output:
Simple concatenate and sort. Duplicates are preserved:
Aw, Kam Foo
Bankov, Peter
Bankov, Peter
Beebe, Ann
Beebe, Ann
El Yassir, Mehdi
Garcia, Debra
Garcia, Hugo
Garcia, Hugo
Giakoumakis, Leo
Gilchrist, Beth
Guy, Wey Yuan
Holm, Michael
Holm, Michael
Liu, Jinghao
McLin, Nkenge
Myrcha, Jacek
Noriega, Fabricio
Potra, Cristina
Toyoshima, Tim
20 total names in list

```

```
Union removes duplicate names:
Aw, Kam Foo
Bankov, Peter
Beebe, Ann
El Yassir, Mehdi
Garcia, Debra
Garcia, Hugo
Giakoumakis, Leo
Gilchrist, Beth
Guy, Wey Yuan
Holm, Michael
Liu, Jinghao
McLin, Nkenge
Myrcha, Jacek
Noriega, Fabricio
Potra, Cristina
Toyoshima, Tim
16 total names in list

Merge based on intersect:
Bankov, Peter
Holm, Michael
Garcia, Hugo
Beebe, Ann
4 total names in list

Concat based on partial name match "Garcia":
Garcia, Debra
Garcia, Hugo
Garcia, Hugo
3 total names in list

*/
```

Compiling the Code

Create a C# console application project, with `using` directives for the System.Linq and System.IO namespaces.

See also

- [LINQ and Strings \(C#\)](#)
- [LINQ and File Directories \(C#\)](#)

How to populate object collections from multiple sources (LINQ) (C#)

12/28/2021 • 4 minutes to read • [Edit Online](#)

This example shows how to merge data from different sources into a sequence of new types.

NOTE

Don't try to join in-memory data or data in the file system with data that is still in a database. Such cross-domain joins can yield undefined results because of different ways in which join operations might be defined for database queries and other types of sources. Additionally, there is a risk that such an operation could cause an out-of-memory exception if the amount of data in the database is large enough. To join data from a database to in-memory data, first call `ToList` or `ToArray` on the database query, and then perform the join on the returned collection.

To create the data file

Copy the names.csv and scores.csv files into your project folder, as described in [How to join content from dissimilar files \(LINQ\) \(C#\)](#).

Example

The following example shows how to use a named type `Student` to store merged data from two in-memory collections of strings that simulate spreadsheet data in .csv format. The first collection of strings represents the student names and IDs, and the second collection represents the student ID (in the first column) and four exam scores. The ID is used as the foreign key.

```
using System;
using System.Collections.Generic;
using System.Linq;

class Student
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public int ID { get; set; }
    public List<int> ExamScores { get; set; }
}

class PopulateCollection
{
    static void Main()
    {
        // These data files are defined in How to join content from
        // dissimilar files (LINQ).

        // Each line of names.csv consists of a last name, a first name, and an
        // ID number, separated by commas. For example, Omelchenko,Svetlana,111
        string[] names = System.IO.File.ReadAllLines(@"../../names.csv");

        // Each line of scores.csv consists of an ID number and four test
        // scores, separated by commas. For example, 111, 97, 92, 81, 60
        string[] scores = System.IO.File.ReadAllLines(@"../../scores.csv");

        // Merge the data sources using a named type.
        // var could be used instead of an explicit type. Note the dynamic
```

```

// var could be used instead of an explicit type. Note the dynamic
// creation of a list of ints for the ExamScores member. The first item
// is skipped in the split string because it is the student ID,
// not an exam score.
IEnumerable<Student> queryNamesScores =
    from nameLine in names
    let splitName = nameLine.Split(',')
    from scoreLine in scores
    let splitScoreLine = scoreLine.Split(',')
    where Convert.ToInt32(splitName[2]) == Convert.ToInt32(splitScoreLine[0])
    select new Student()
    {
        FirstName = splitName[0],
        LastName = splitName[1],
        ID = Convert.ToInt32(splitName[2]),
        ExamScores = (from scoreAsText in splitScoreLine.Skip(1)
                      select Convert.ToInt32(scoreAsText)).
                      ToList()
    };

// Optional. Store the newly created student objects in memory
// for faster access in future queries. This could be useful with
// very large data files.
List<Student> students = queryNamesScores.ToList();

// Display each student's name and exam score average.
foreach (var student in students)
{
    Console.WriteLine("The average score of {0} {1} is {2}.",
        student.FirstName, student.LastName,
        student.ExamScores.Average());
}

//Keep console window open in debug mode
Console.WriteLine("Press any key to exit.");
Console.ReadKey();
}
}
/* Output:
    The average score of Omelchenko Svetlana is 82.5.
    The average score of O'Donnell Claire is 72.25.
    The average score of Mortensen Sven is 84.5.
    The average score of Garcia Cesar is 88.25.
    The average score of Garcia Debra is 67.
    The average score of Fakhouri Fadi is 92.25.
    The average score of Feng Hanying is 88.
    The average score of Garcia Hugo is 85.75.
    The average score of Tucker Lance is 81.75.
    The average score of Adams Terry is 85.25.
    The average score of Zabokritski Eugene is 83.
    The average score of Tucker Michael is 92.
*/

```

In the [select](#) clause, an object initializer is used to instantiate each new `Student` object by using the data from the two sources.

If you don't have to store the results of a query, anonymous types can be more convenient than named types. Named types are required if you pass the query results outside the method in which the query is executed. The following example executes the same task as the previous example, but uses anonymous types instead of named types:

```

// Merge the data sources by using an anonymous type.
// Note the dynamic creation of a list of ints for the
// ExamScores member. We skip 1 because the first string
// in the array is the student ID, not an exam score.
var queryNamesScores2 =
    from nameLine in names
    let splitName = nameLine.Split(',')
    from scoreLine in scores
    let splitScoreLine = scoreLine.Split(',')
    where Convert.ToInt32(splitName[2]) == Convert.ToInt32(splitScoreLine[0])
    select new
    {
        First = splitName[0],
        Last = splitName[1],
        ExamScores = (from scoreAsText in splitScoreLine.Skip(1)
                      select Convert.ToInt32(scoreAsText))
                      .ToList()
    };

// Display each student's name and exam score average.
foreach (var student in queryNamesScores2)
{
    Console.WriteLine("The average score of {0} {1} is {2}.",
        student.First, student.Last, student.ExamScores.Average());
}

```

See also

- [LINQ and Strings \(C#\)](#)
- [Object and Collection Initializers](#)
- [Anonymous Types](#)

How to split a file into many files by using groups (LINQ) (C#)

12/28/2021 • 2 minutes to read • [Edit Online](#)

This example shows one way to merge the contents of two files and then create a set of new files that organize the data in a new way.

To create the data files

1. Copy these names into a text file that is named names1.txt and save it in your project folder:

```
Bankov, Peter
Holm, Michael
Garcia, Hugo
Potra, Cristina
Noriega, Fabricio
Aw, Kam Foo
Beebe, Ann
Toyoshima, Tim
Guy, Wey Yuan
Garcia, Debra
```

2. Copy these names into a text file that is named names2.txt and save it in your project folder: Note that the two files have some names in common.

```
Liu, Jinghao
Bankov, Peter
Holm, Michael
Garcia, Hugo
Beebe, Ann
Gilchrist, Beth
Myrcha, Jacek
Giakoumakis, Leo
McLin, Nkenge
El Yassir, Mehdi
```

Example

```
class SplitWithGroups
{
    static void Main()
    {
        string[] fileA = System.IO.File.ReadAllLines(@"../../names1.txt");
        string[] fileB = System.IO.File.ReadAllLines(@"../../names2.txt");

        // Concatenate and remove duplicate names based on
        // default string comparer
        var mergeQuery = fileA.Union(fileB);

        // Group the names by the first letter in the last name.
        var groupQuery = from name in mergeQuery
                        let n = name.Split(',')
                        group name by n[0][0] into g
                        orderby g.Key
                        select g;
    }
}
```



```

// Create a new file for each group that was created
// Note that nested foreach loops are required to access
// individual items with each group.
foreach (var g in groupQuery)
{
    // Create the new file name.
    string fileName = @"../.././testFile_" + g.Key + ".txt";

    // Output to display.
    Console.WriteLine(g.Key);

    // Write file.
    using (System.IO.StreamWriter sw = new System.IO.StreamWriter(fileName))
    {
        foreach (var item in g)
        {
            sw.WriteLine(item);
            // Output to console for example purposes.
            Console.WriteLine("  {0}", item);
        }
    }
    // Keep console window open in debug mode.
    Console.WriteLine("Files have been written. Press any key to exit");
    Console.ReadKey();
}
}
/* Output:
A
  Aw, Kam Foo
B
  Bankov, Peter
  Beebe, Ann
E
  El Yassir, Mehdi
G
  Garcia, Hugo
  Guy, Wey Yuan
  Garcia, Debra
  Gilchrist, Beth
  Giakoumakis, Leo
H
  Holm, Michael
L
  Liu, Jinghao
M
  Myrcha, Jacek
  McLin, Nkenge
N
  Noriega, Fabricio
P
  Potra, Cristina
T
  Toyoshima, Tim
*/

```

The program writes a separate file for each group in the same folder as the data files.

Compiling the Code

Create a C# console application project, with `using` directives for the System.Linq and System.IO namespaces.

See also

- [LINQ and Strings \(C#\)](#)

- [LINQ and File Directories \(C#\)](#)

How to join content from dissimilar files (LINQ) (C#)

12/28/2021 • 2 minutes to read • [Edit Online](#)

This example shows how to join data from two comma-delimited files that share a common value that is used as a matching key. This technique can be useful if you have to combine data from two spreadsheets, or from a spreadsheet and from a file that has another format, into a new file. You can modify the example to work with any kind of structured text.

To create the data files

1. Copy the following lines into a file that is named *scores.csv* and save it to your project folder. The file represents spreadsheet data. Column 1 is the student's ID, and columns 2 through 5 are test scores.

```
111, 97, 92, 81, 60
112, 75, 84, 91, 39
113, 88, 94, 65, 91
114, 97, 89, 85, 82
115, 35, 72, 91, 70
116, 99, 86, 90, 94
117, 93, 92, 80, 87
118, 92, 90, 83, 78
119, 68, 79, 88, 92
120, 99, 82, 81, 79
121, 96, 85, 91, 60
122, 94, 92, 91, 91
```

2. Copy the following lines into a file that is named *names.csv* and save it to your project folder. The file represents a spreadsheet that contains the student's last name, first name, and student ID.

```
Omelchenko,Svetlana,111
O'Donnell,Claire,112
Mortensen,Sven,113
Garcia,Cesar,114
Garcia,Debra,115
Fakhouri,Fadi,116
Feng,Hanying,117
Garcia,Hugo,118
Tucker,Lance,119
Adams,Terry,120
Zabokritski,Eugene,121
Tucker,Michael,122
```

Example

```
using System;
using System.Collections.Generic;
using System.Linq;

class JoinStrings
{
    static void Main()
    {
        // Join content from dissimilar files that contain
        // related information. File names.csv contains the student
        // name plus an ID number. File scores.csv contains the ID
```

```

// and a set of four test scores. The following query joins
// the scores to the student names by using ID as a
// matching key.

string[] names = System.IO.File.ReadAllLines(@"../../names.csv");
string[] scores = System.IO.File.ReadAllLines(@"../../scores.csv");

// Name:      Last[0],      First[1],  ID[2]
//           Omelchenko,    Svetlana,  11
// Score:      StudentID[0], Exam1[1]   Exam2[2], Exam3[3], Exam4[4]
//           111,          97,          92,      81,      60

// This query joins two dissimilar spreadsheets based on common ID value.
// Multiple from clauses are used instead of a join clause
// in order to store results of id.Split.
IEnumerable<string> scoreQuery1 =
    from name in names
    let nameFields = name.Split(',')
    from id in scores
    let scoreFields = id.Split(',')
    where Convert.ToInt32(nameFields[2]) == Convert.ToInt32(scoreFields[0])
    select nameFields[0] + "," + scoreFields[1] + "," + scoreFields[2]
        + "," + scoreFields[3] + "," + scoreFields[4];

// Pass a query variable to a method and execute it
// in the method. The query itself is unchanged.
OutputQueryResults(scoreQuery1, "Merge two spreadsheets:");

// Keep console window open in debug mode.
Console.WriteLine("Press any key to exit");
Console.ReadKey();
}

static void OutputQueryResults(IEnumerable<string> query, string message)
{
    Console.WriteLine(System.Environment.NewLine + message);
    foreach (string item in query)
    {
        Console.WriteLine(item);
    }
    Console.WriteLine("{0} total names in list", query.Count());
}
}
/* Output:
Merge two spreadsheets:
Omelchenko, 97, 92, 81, 60
O'Donnell, 75, 84, 91, 39
Mortensen, 88, 94, 65, 91
Garcia, 97, 89, 85, 82
Garcia, 35, 72, 91, 70
Fakhouri, 99, 86, 90, 94
Feng, 93, 92, 80, 87
Garcia, 92, 90, 83, 78
Tucker, 68, 79, 88, 92
Adams, 99, 82, 81, 79
Zabokritski, 96, 85, 91, 60
Tucker, 94, 92, 91, 91
12 total names in list
*/

```

See also

- [LINQ and Strings \(C#\)](#)
- [LINQ and File Directories \(C#\)](#)

How to compute column values in a CSV text file (LINQ) (C#)

12/28/2021 • 3 minutes to read • [Edit Online](#)

This example shows how to perform aggregate computations such as Sum, Average, Min, and Max on the columns of a .csv file. The example principles that are shown here can be applied to other types of structured text.

To create the source file

1. Copy the following lines into a file that is named scores.csv and save it in your project folder. Assume that the first column represents a student ID, and subsequent columns represent scores from four exams.

```
111, 97, 92, 81, 60
112, 75, 84, 91, 39
113, 88, 94, 65, 91
114, 97, 89, 85, 82
115, 35, 72, 91, 70
116, 99, 86, 90, 94
117, 93, 92, 80, 87
118, 92, 90, 83, 78
119, 68, 79, 88, 92
120, 99, 82, 81, 79
121, 96, 85, 91, 60
122, 94, 92, 91, 91
```

Example

```
class SumColumns
{
    static void Main(string[] args)
    {
        string[] lines = System.IO.File.ReadAllLines(@"../.././scores.csv");

        // Specifies the column to compute.
        int exam = 3;

        // Spreadsheet format:
        // Student ID   Exam#1  Exam#2  Exam#3  Exam#4
        // 111,         97,     92,      81,     60

        // Add one to exam to skip over the first column,
        // which holds the student ID.
        SingleColumn(lines, exam + 1);
        Console.WriteLine();
        MultiColumns(lines);

        Console.WriteLine("Press any key to exit");
        Console.ReadKey();
    }

    static void SingleColumn(IEnumerable<string> strs, int examNum)
    {
        Console.WriteLine("Single Column Query:");

        // Parameter examNum specifies the column to
```

```

// run the calculations on. This value could be
// passed in dynamically at run time.

// Variable columnQuery is an IEnumerable<int>.
// The following query performs two steps:
// 1) use Split to break each row (a string) into an array
//    of strings,
// 2) convert the element at position examNum to an int
//    and select it.
var columnQuery =
    from line in strs
    let elements = line.Split(',')
    select Convert.ToInt32(elements[examNum]);

// Execute the query and cache the results to improve
// performance. This is helpful only with very large files.
var results = columnQuery.ToList();

// Perform aggregate calculations Average, Max, and
// Min on the column specified by examNum.
double average = results.Average();
int max = results.Max();
int min = results.Min();

Console.WriteLine("Exam #{0}: Average:{1:##.##} High Score:{2} Low Score:{3}",
    examNum, average, max, min);
}

static void MultiColumns(IEnumerable<string> strs)
{
    Console.WriteLine("Multi Column Query:");

    // Create a query, multiColQuery. Explicit typing is used
    // to make clear that, when executed, multiColQuery produces
    // nested sequences. However, you get the same results by
    // using 'var'.

    // The multiColQuery query performs the following steps:
    // 1) use Split to break each row (a string) into an array
    //    of strings,
    // 2) use Skip to skip the "Student ID" column, and store the
    //    rest of the row in scores.
    // 3) convert each score in the current row from a string to
    //    an int, and select that entire sequence as one row
    //    in the results.
    IEnumerable<IEnumerable<int>> multiColQuery =
        from line in strs
        let elements = line.Split(',')
        let scores = elements.Skip(1)
        select (from str in scores
            select Convert.ToInt32(str));

    // Execute the query and cache the results to improve
    // performance.
    // ToArray could be used instead of ToList.
    var results = multiColQuery.ToList();

    // Find out how many columns you have in results.
    int columnCount = results[0].Count();

    // Perform aggregate calculations Average, Max, and
    // Min on each column.
    // Perform one iteration of the loop for each column
    // of scores.
    // You can use a for loop instead of a foreach loop
    // because you already executed the multiColQuery
    // query by calling ToList.
    for (int column = 0; column < columnCount; column++)
    {

```

```

        var results2 = from row in results
                        select row.ElementAt(column);
        double average = results2.Average();
        int max = results2.Max();
        int min = results2.Min();

        // Add one to column because the first exam is Exam #1,
        // not Exam #0.
        Console.WriteLine("Exam #{0} Average: {1:##.##} High Score: {2} Low Score: {3}",
                           column + 1, average, max, min);
    }
}
/* Output:
    Single Column Query:
    Exam #4: Average:76.92 High Score:94 Low Score:39

    Multi Column Query:
    Exam #1 Average: 86.08 High Score: 99 Low Score: 35
    Exam #2 Average: 86.42 High Score: 94 Low Score: 72
    Exam #3 Average: 84.75 High Score: 91 Low Score: 65
    Exam #4 Average: 76.92 High Score: 94 Low Score: 39
*/

```

The query works by using the [Split](#) method to convert each line of text into an array. Each array element represents a column. Finally, the text in each column is converted to its numeric representation. If your file is a tab-separated file, just update the argument in the `Split` method to `\t`.

Compiling the Code

Create a C# console application project, with `using` directives for the `System.Linq` and `System.IO` namespaces.

See also

- [LINQ and Strings \(C#\)](#)
- [LINQ and File Directories \(C#\)](#)

How to query an assembly's metadata with Reflection (LINQ) (C#)

12/28/2021 • 2 minutes to read • [Edit Online](#)

The .NET reflection APIs can be used to examine the metadata in a .NET assembly and create collections of types, type members, parameters, and so on that are in that assembly. Because these collections support the generic [IEnumerable<T>](#) interface, they can be queried by using LINQ.

The following example shows how LINQ can be used with reflection to retrieve specific metadata about methods that match a specified search criterion. In this case, the query will find the names of all the methods in the assembly that return enumerable types such as arrays.

Example

```
using System;
using System.Linq;
using System.Reflection;

class ReflectionHowTO
{
    static void Main()
    {
        Assembly assembly = Assembly.Load("System.Core, Version=3.5.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089");
        var pubTypesQuery = from type in assembly.GetTypes()
                            where type.IsPublic
                                from method in type.GetMethods()
                                where method.ReturnType.IsArray == true
                                    || ( method.ReturnType.GetInterface(
                                        typeof(System.Collections.Generic.IEnumerable<>).FullName ) != null
                                        && method.ReturnType.FullName != "System.String" )
                                group method.ToString() by type.ToString();

        foreach (var groupOfMethods in pubTypesQuery)
        {
            Console.WriteLine("Type: {0}", groupOfMethods.Key);
            foreach (var method in groupOfMethods)
            {
                Console.WriteLine("  {0}", method);
            }
        }

        Console.WriteLine("Press any key to exit... ");
        Console.ReadKey();
    }
}
```

The example uses the [Assembly.GetTypes](#) method to return an array of types in the specified assembly. The [where](#) filter is applied so that only public types are returned. For each public type, a subquery is generated by using the [MethodInfo](#) array that is returned from the [Type.GetMethods](#) call. These results are filtered to return only those methods whose return type is an array or else a type that implements [IEnumerable<T>](#). Finally, these results are grouped by using the type name as a key.

See also

- [LINQ to Objects \(C#\)](#)

How to query an assembly's metadata with Reflection (LINQ) (C#)

12/28/2021 • 2 minutes to read • [Edit Online](#)

The .NET reflection APIs can be used to examine the metadata in a .NET assembly and create collections of types, type members, parameters, and so on that are in that assembly. Because these collections support the generic [IEnumerable<T>](#) interface, they can be queried by using LINQ.

The following example shows how LINQ can be used with reflection to retrieve specific metadata about methods that match a specified search criterion. In this case, the query will find the names of all the methods in the assembly that return enumerable types such as arrays.

Example

```
using System;
using System.Linq;
using System.Reflection;

class ReflectionHowTO
{
    static void Main()
    {
        Assembly assembly = Assembly.Load("System.Core, Version=3.5.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089");
        var pubTypesQuery = from type in assembly.GetTypes()
                            where type.IsPublic
                                from method in type.GetMethods()
                                where method.ReturnType.IsArray == true
                                    || ( method.ReturnType.GetInterface(
                                        typeof(System.Collections.Generic.IEnumerable<>).FullName ) != null
                                    && method.ReturnType.FullName != "System.String" )
                                group method.ToString() by type.ToString();

        foreach (var groupOfMethods in pubTypesQuery)
        {
            Console.WriteLine("Type: {0}", groupOfMethods.Key);
            foreach (var method in groupOfMethods)
            {
                Console.WriteLine("  {0}", method);
            }
        }

        Console.WriteLine("Press any key to exit... ");
        Console.ReadKey();
    }
}
```

The example uses the [Assembly.GetTypes](#) method to return an array of types in the specified assembly. The [where](#) filter is applied so that only public types are returned. For each public type, a subquery is generated by using the [MethodInfo](#) array that is returned from the [Type.GetMethods](#) call. These results are filtered to return only those methods whose return type is an array or else a type that implements [IEnumerable<T>](#). Finally, these results are grouped by using the type name as a key.

See also

- [LINQ to Objects \(C#\)](#)

LINQ and file directories (C#)

12/28/2021 • 2 minutes to read • [Edit Online](#)

Many file system operations are essentially queries and are therefore well suited to the LINQ approach.

The queries in this section are non-destructive. They are not used to change the contents of the original files or folders. This follows the rule that queries should not cause any side-effects. In general, any code (including queries that perform create / update / delete operators) that modifies source data should be kept separate from the code that just queries the data.

This section contains the following topics:

[How to query for files with a specified attribute or name \(C#\)](#)

Shows how to search for files by examining one or more properties of its [FileInfo](#) object.

[How to group files by extension \(LINQ\) \(C#\)](#)

Shows how to return groups of [FileInfo](#) object based on their file name extension.

[How to query for the total number of bytes in a set of folders \(LINQ\) \(C#\)](#)

Shows how to return the total number of bytes in all the files in a specified directory tree.

[How to compare the contents of two folders \(LINQ\) \(C#\)](#)

Shows how to return all the files that are present in two specified folders, and also all the files that are present in one folder but not the other.

[How to query for the largest file or files in a directory tree \(LINQ\) \(C#\)](#)

Shows how to return the largest or smallest file, or a specified number of files, in a directory tree.

[How to query for duplicate files in a directory tree \(LINQ\) \(C#\)](#)

Shows how to group for all file names that occur in more than one location in a specified directory tree. Also shows how to perform more complex comparisons based on a custom comparer.

[How to query the contents of files in a folder \(LINQ\) \(C#\)](#)

Shows how to iterate through folders in a tree, open each file, and query the file's contents.

Comments

There is some complexity involved in creating a data source that accurately represents the contents of the file system and handles exceptions gracefully. The examples in this section create a snapshot collection of [FileInfo](#) objects that represents all the files under a specified root folder and all its subfolders. The actual state of each [FileInfo](#) may change in the time between when you begin and end executing a query. For example, you can create a list of [FileInfo](#) objects to use as a data source. If you try to access the `Length` property in a query, the [FileInfo](#) object will try to access the file system to update the value of `Length`. If the file no longer exists, you will get a [FileNotFoundException](#) in your query, even though you are not querying the file system directly. Some queries in this section use a separate method that consumes these particular exceptions in certain cases. Another option is to keep your data source updated dynamically by using the [FileSystemWatcher](#).

See also

- [LINQ to Objects \(C#\)](#)

How to query for files with a specified attribute or name (C#)

12/28/2021 • 2 minutes to read • [Edit Online](#)

This example shows how to find all files that have a specified file name extension (for example ".txt") in a specified directory tree. It also shows how to return either the newest or oldest file in the tree based on the creation time.

Example

```
class FindFileByExtension
{
    // This query will produce the full path for all .txt files
    // under the specified folder including subfolders.
    // It orders the list according to the file name.
    static void Main()
    {
        string startFolder = @"c:\program files\Microsoft Visual Studio 9.0\";

        // Take a snapshot of the file system.
        System.IO.DirectoryInfo dir = new System.IO.DirectoryInfo(startFolder);

        // This method assumes that the application has discovery permissions
        // for all folders under the specified path.
        IEnumerable<System.IO.FileInfo> fileList = dir.GetFiles("*.*",
            System.IO.SearchOption.AllDirectories);

        //Create the query
        IEnumerable<System.IO.FileInfo> fileQuery =
            from file in fileList
            where file.Extension == ".txt"
            orderby file.Name
            select file;

        //Execute the query. This might write out a lot of files!
        foreach (System.IO.FileInfo fi in fileQuery)
        {
            Console.WriteLine(fi.FullName);
        }

        // Create and execute a new query by using the previous
        // query as a starting point. fileQuery is not
        // executed again until the call to Last()
        var newestFile =
            (from file in fileQuery
             orderby file.CreationTime
             select new { file.FullName, file.CreationTime })
            .Last();

        Console.WriteLine("\r\nThe newest .txt file is {0}. Creation time: {1}",
            newestFile.FullName, newestFile.CreationTime);

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit");
        Console.ReadKey();
    }
}
```

Compiling the Code

Create a C# console application project, with `using` directives for the System.Linq and System.IO namespaces.

See also

- [LINQ to Objects \(C#\)](#)
- [LINQ and File Directories \(C#\)](#)

How to group files by extension (LINQ) (C#)

12/28/2021 • 2 minutes to read • [Edit Online](#)

This example shows how LINQ can be used to perform advanced grouping and sorting operations on lists of files or folders. It also shows how to page output in the console window by using the [Skip](#) and [Take](#) methods.

Example

The following query shows how to group the contents of a specified directory tree by the file name extension.

```
class GroupByExtension
{
    // This query will sort all the files under the specified folder
    // and subfolder into groups keyed by the file extension.
    private static void Main()
    {
        // Take a snapshot of the file system.
        string startFolder = @"c:\program files\Microsoft Visual Studio 9.0\Common7";

        // Used in WriteLine to trim output lines.
        int trimLength = startFolder.Length;

        // Take a snapshot of the file system.
        System.IO.DirectoryInfo dir = new System.IO.DirectoryInfo(startFolder);

        // This method assumes that the application has discovery permissions
        // for all folders under the specified path.
        IEnumerable<System.IO.FileInfo> fileList = dir.GetFiles("*.*",
            System.IO.SearchOption.AllDirectories);

        // Create the query.
        var queryGroupByExt =
            from file in fileList
            group file by file.Extension.ToLower() into fileGroup
            orderby fileGroup.Key
            select fileGroup;

        // Display one group at a time. If the number of
        // entries is greater than the number of lines
        // in the console window, then page the output.
        PageOutput(trimLength, queryGroupByExt);
    }

    // This method specifically handles group queries of FileInfo objects with string keys.
    // It can be modified to work for any long listings of data. Note that explicit typing
    // must be used in method signatures. The groupbyExtList parameter is a query that produces
    // groups of FileInfo objects with string keys.
    private static void PageOutput(int rootLength,
                                    IEnumerable<System.Linq.IGrouping<string, System.IO.FileInfo>>
groupByExtList)
    {
        // Flag to break out of paging loop.
        bool goAgain = true;

        // "3" = 1 line for extension + 1 for "Press any key" + 1 for input cursor.
        int numLines = Console.WindowHeight - 3;

        // Iterate through the outer collection of groups.
        foreach (var filegroup in groupByExtList)
        {
            // Start a new extension at the top of a page.
```

```

int currentLine = 0;

// Output only as many lines of the current group as will fit in the window.
do
{
    Console.Clear();
    Console.WriteLine(filegroup.Key == String.Empty ? "[none]" : filegroup.Key);

    // Get 'numLines' number of items starting at number 'currentLine'.
    var resultPage = filegroup.Skip(currentLine).Take(numLines);

    //Execute the resultPage query
    foreach (var f in resultPage)
    {
        Console.WriteLine("\t{0}", f.FullName.Substring(rootLength));
    }

    // Increment the line counter.
    currentLine += numLines;

    // Give the user a chance to escape.
    Console.WriteLine("Press any key to continue or the 'End' key to break...");
    ConsoleKey key = Console.ReadKey().Key;
    if (key == ConsoleKey.End)
    {
        goAgain = false;
        break;
    }
} while (currentLine < filegroup.Count());

if (goAgain == false)
    break;
}
}
}

```

The output from this program can be long, depending on the details of the local file system and what the `startFolder` is set to. To enable viewing of all results, this example shows how to page through results. The same techniques can be applied to Windows and Web applications. Notice that because the code pages the items in a group, a nested `foreach` loop is required. There is also some additional logic to compute the current position in the list, and to enable the user to stop paging and exit the program. In this particular case, the paging query is run against the cached results from the original query. In other contexts, such as LINQ to SQL, such caching is not required.

Compiling the Code

Create a C# console application project, with `using` directives for the `System.Linq` and `System.IO` namespaces.

See also

- [LINQ to Objects \(C#\)](#)
- [LINQ and File Directories \(C#\)](#)

How to query for the total number of bytes in a set of folders (LINQ) (C#)

12/28/2021 • 2 minutes to read • [Edit Online](#)

This example shows how to retrieve the total number of bytes used by all the files in a specified folder and all its subfolders.

Example

The [Sum](#) method adds the values of all the items selected in the `select` clause. You can easily modify this query to retrieve the biggest or smallest file in the specified directory tree by calling the [Min](#) or [Max](#) method instead of [Sum](#).

```

class QuerySize
{
    public static void Main()
    {
        string startFolder = @"c:\program files\Microsoft Visual Studio 9.0\VC#";

        // Take a snapshot of the file system.
        // This method assumes that the application has discovery permissions
        // for all folders under the specified path.
        IEnumerable<string> fileList = System.IO.Directory.GetFiles(startFolder, "**.*",
System.IO.SearchOption.AllDirectories);

        var fileQuery = from file in fileList
                        select GetFileLength(file);

        // Cache the results to avoid multiple trips to the file system.
        long[] fileLengths = fileQuery.ToArray();

        // Return the size of the largest file
        long largestFile = fileLengths.Max();

        // Return the total number of bytes in all the files under the specified folder.
        long totalBytes = fileLengths.Sum();

        Console.WriteLine("There are {0} bytes in {1} files under {2}",
            totalBytes, fileList.Count(), startFolder);
        Console.WriteLine("The largest files is {0} bytes.", largestFile);

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }

    // This method is used to swallow the possible exception
    // that can be raised when accessing the System.IO.FileInfo.Length property.
    static long GetFileLength(string filename)
    {
        long retval;
        try
        {
            System.IO.FileInfo fi = new System.IO.FileInfo(filename);
            retval = fi.Length;
        }
        catch (System.IO.FileNotFoundException)
        {
            // If a file is no longer present,
            // just add zero bytes to the total.
            retval = 0;
        }
        return retval;
    }
}

```

If you only have to count the number of bytes in a specified directory tree, you can do this more efficiently without creating a LINQ query, which incurs the overhead of creating the list collection as a data source. The usefulness of the LINQ approach increases as the query becomes more complex, or when you have to run multiple queries against the same data source.

The query calls out to a separate method to obtain the file length. It does this in order to consume the possible exception that will be raised if the file was deleted on another thread after the [FileInfo](#) object was created in the call to `GetFiles`. Even though the [FileInfo](#) object has already been created, the exception can occur because a [FileInfo](#) object will try to refresh its [Length](#) property with the most current length the first time the property is accessed. By putting this operation in a try-catch block outside the query, the code follows the rule of avoiding operations in queries that can cause side-effects. In general, great care must be taken when you consume

exceptions to make sure that an application is not left in an unknown state.

Compiling the Code

Create a C# console application project, with `using` directives for the System.Linq and System.IO namespaces.

See also

- [LINQ to Objects \(C#\)](#)
- [LINQ and File Directories \(C#\)](#)

How to compare the contents of two folders (LINQ) (C#)

12/28/2021 • 2 minutes to read • [Edit Online](#)

This example demonstrates three ways to compare two file listings:

- By querying for a Boolean value that specifies whether the two file lists are identical.
- By querying for the intersection to retrieve the files that are in both folders.
- By querying for the set difference to retrieve the files that are in one folder but not the other.

NOTE

The techniques shown here can be adapted to compare sequences of objects of any type.

The `FileComparer` class shown here demonstrates how to use a custom comparer class together with the Standard Query Operators. The class is not intended for use in real-world scenarios. It just uses the name and length in bytes of each file to determine whether the contents of each folder are identical or not. In a real-world scenario, you should modify this comparer to perform a more rigorous equality check.

Example

```
namespace QueryCompareTwoDirs
{
    class CompareDirs
    {
        static void Main(string[] args)
        {
            // Create two identical or different temporary folders
            // on a local drive and change these file paths.
            string pathA = @"C:\TestDir";
            string pathB = @"C:\TestDir2";

            System.IO.DirectoryInfo dir1 = new System.IO.DirectoryInfo(pathA);
            System.IO.DirectoryInfo dir2 = new System.IO.DirectoryInfo(pathB);

            // Take a snapshot of the file system.
            IEnumerable<System.IO.FileInfo> list1 = dir1.GetFiles("*.*",
System.IO.SearchOption.AllDirectories);
            IEnumerable<System.IO.FileInfo> list2 = dir2.GetFiles("*.*",
System.IO.SearchOption.AllDirectories);

            //A custom file comparer defined below
            FileCompare myFileCompare = new FileCompare();

            // This query determines whether the two folders contain
            // identical file lists, based on the custom file comparer
            // that is defined in the FileCompare class.
            // The query executes immediately because it returns a bool.
            bool areIdentical = list1.SequenceEqual(list2, myFileCompare);

            if (areIdentical == true)
            {
                Console.WriteLine("the two folders are the same");
            }
        }
    }
}
```

```

        Console.WriteLine("The two folders are the same");
    }
    else
    {
        Console.WriteLine("The two folders are not the same");
    }

    // Find the common files. It produces a sequence and doesn't
    // execute until the foreach statement.
    var queryCommonFiles = list1.Intersect(list2, myFileCompare);

    if (queryCommonFiles.Any())
    {
        Console.WriteLine("The following files are in both folders:");
        foreach (var v in queryCommonFiles)
        {
            Console.WriteLine(v.FullName); //shows which items end up in result list
        }
    }
    else
    {
        Console.WriteLine("There are no common files in the two folders.");
    }

    // Find the set difference between the two folders.
    // For this example we only check one way.
    var queryList1Only = (from file in list1
                          select file).Except(list2, myFileCompare);

    Console.WriteLine("The following files are in list1 but not list2:");
    foreach (var v in queryList1Only)
    {
        Console.WriteLine(v.FullName);
    }

    // Keep the console window open in debug mode.
    Console.WriteLine("Press any key to exit.");
    Console.ReadKey();
}
}

// This implementation defines a very simple comparison
// between two FileInfo objects. It only compares the name
// of the files being compared and their length in bytes.
class FileCompare : System.Collections.Generic.IEqualityComparer<System.IO.FileInfo>
{
    public FileCompare() { }

    public bool Equals(System.IO.FileInfo f1, System.IO.FileInfo f2)
    {
        return (f1.Name == f2.Name &&
                f1.Length == f2.Length);
    }

    // Return a hash that reflects the comparison criteria. According to the
    // rules for IEqualityComparer<T>, if Equals is true, then the hash codes must
    // also be equal. Because equality as defined here is a simple value equality, not
    // reference identity, it is possible that two or more objects will produce the same
    // hash code.
    public int GetHashCode(System.IO.FileInfo fi)
    {
        string s = $"{fi.Name}{fi.Length}";
        return s.GetHashCode();
    }
}
}

```

Compiling the Code

Create a C# console application project, with `using` directives for the System.Linq and System.IO namespaces.

See also

- [LINQ to Objects \(C#\)](#)
- [LINQ and File Directories \(C#\)](#)

How to query for the largest file or files in a directory tree (LINQ) (C#)

12/28/2021 • 3 minutes to read • [Edit Online](#)

This example shows five queries related to file size in bytes:

- How to retrieve the size in bytes of the largest file.
- How to retrieve the size in bytes of the smallest file.
- How to retrieve the [FileInfo](#) object largest or smallest file from one or more folders under a specified root folder.
- How to retrieve a sequence such as the 10 largest files.
- How to order files into groups based on their file size in bytes, ignoring files that are less than a specified size.

Example

The following example contains five separate queries that show how to query and group files, depending on their file size in bytes. You can easily modify these examples to base the query on some other property of the [FileInfo](#) object.

```
class QueryBySize
{
    static void Main(string[] args)
    {
        QueryFilesBySize();
        Console.WriteLine("Press any key to exit");
        Console.ReadKey();
    }

    private static void QueryFilesBySize()
    {
        string startFolder = @"c:\program files\Microsoft Visual Studio 9.0\";

        // Take a snapshot of the file system.
        System.IO.DirectoryInfo dir = new System.IO.DirectoryInfo(startFolder);

        // This method assumes that the application has discovery permissions
        // for all folders under the specified path.
        IEnumerable<System.IO.FileInfo> fileList = dir.GetFiles("*.*",
            System.IO.SearchOption.AllDirectories);

        //Return the size of the largest file
        long maxSize =
            (from file in fileList
             let len = GetFileLength(file)
             select len)
            .Max();

        Console.WriteLine("The length of the largest file under {0} is {1}",
            startFolder, maxSize);

        // Return the FileInfo object for the largest file
        // by sorting and selecting from beginning of list
        System.IO.FileInfo longestFile =
```

```

        (from file in fileList
         let len = GetFileLength(file)
         where len > 0
         orderby len descending
         select file)
        .First();

Console.WriteLine("The largest file under {0} is {1} with a length of {2} bytes",
                  startFolder, longestFile.FullName, longestFile.Length);

//Return the FileInfo of the smallest file
System.IO.FileInfo smallestFile =
    (from file in fileList
     let len = GetFileLength(file)
     where len > 0
     orderby len ascending
     select file).First();

Console.WriteLine("The smallest file under {0} is {1} with a length of {2} bytes",
                  startFolder, smallestFile.FullName, smallestFile.Length);

//Return the FileInfos for the 10 largest files
// queryTenLargest is an IEnumerable<System.IO.FileInfo>
var queryTenLargest =
    (from file in fileList
     let len = GetFileLength(file)
     orderby len descending
     select file).Take(10);

Console.WriteLine("The 10 largest files under {0} are:", startFolder);

foreach (var v in queryTenLargest)
{
    Console.WriteLine("{0}: {1} bytes", v.FullName, v.Length);
}

// Group the files according to their size, leaving out
// files that are less than 200000 bytes.
var querySizeGroups =
    from file in fileList
    let len = GetFileLength(file)
    where len > 0
    group file by (len / 100000) into fileGroup
    where fileGroup.Key >= 2
    orderby fileGroup.Key descending
    select fileGroup;

foreach (var filegroup in querySizeGroups)
{
    Console.WriteLine(filegroup.Key.ToString() + "00000");
    foreach (var item in filegroup)
    {
        Console.WriteLine("\t{0}: {1}", item.Name, item.Length);
    }
}
}

// This method is used to swallow the possible exception
// that can be raised when accessing the FileInfo.Length property.
// In this particular case, it is safe to swallow the exception.
static long GetFileLength(System.IO.FileInfo fi)
{
    long retVal;
    try
    {
        retVal = fi.Length;
    }
    catch (System.IO.FileNotFoundException)
    {
    }
}

```



```
        // If a file is no longer present,  
        // just add zero bytes to the total.  
        retval = 0;  
    }  
    return retval;  
}  
  
}
```

To return one or more complete [FileInfo](#) objects, the query first must examine each one in the data source, and then sort them by the value of their `Length` property. Then it can return the single one or the sequence with the greatest lengths. Use [First](#) to return the first element in a list. Use [Take](#) to return the first n number of elements. Specify a descending sort order to put the smallest elements at the start of the list.

The query calls out to a separate method to obtain the file size in bytes in order to consume the possible exception that will be raised in the case where a file was deleted on another thread in the time period since the [FileInfo](#) object was created in the call to `GetFiles`. Even though the [FileInfo](#) object has already been created, the exception can occur because a [FileInfo](#) object will try to refresh its `Length` property by using the most current size in bytes the first time the property is accessed. By putting this operation in a try-catch block outside the query, we follow the rule of avoiding operations in queries that can cause side-effects. In general, great care must be taken when consuming exceptions, to make sure that an application is not left in an unknown state.

Compiling the Code

Create a C# console application project, with `using` directives for the `System.Linq` and `System.IO` namespaces.

See also

- [LINQ to Objects \(C#\)](#)
- [LINQ and File Directories \(C#\)](#)

How to query for duplicate files in a directory tree (LINQ) (C#)

12/28/2021 • 3 minutes to read • [Edit Online](#)

Sometimes files that have the same name may be located in more than one folder. For example, under the Visual Studio installation folder, several folders have a readme.htm file. This example shows how to query for such duplicate file names under a specified root folder. The second example shows how to query for files whose size and LastWrite times also match.

Example

```
class QueryDuplicateFileNames
{
    static void Main(string[] args)
    {
        // Uncomment QueryDuplicates2 to run that query.
        QueryDuplicates();
        // QueryDuplicates2();

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }

    static void QueryDuplicates()
    {
        // Change the root drive or folder if necessary
        string startFolder = @"c:\program files\Microsoft Visual Studio 9.0\";

        // Take a snapshot of the file system.
        System.IO.DirectoryInfo dir = new System.IO.DirectoryInfo(startFolder);

        // This method assumes that the application has discovery permissions
        // for all folders under the specified path.
        IEnumerable<System.IO.FileInfo> fileList = dir.GetFiles("*.*",
            System.IO.SearchOption.AllDirectories);

        // used in WriteLine to keep the lines shorter
        int charsToSkip = startFolder.Length;

        // var can be used for convenience with groups.
        var queryDupNames =
            from file in fileList
            group file.FullName.Substring(charsToSkip) by file.Name into fileGroup
            where fileGroup.Count() > 1
            select fileGroup;

        // Pass the query to a method that will
        // output one page at a time.
        PageOutput<string, string>(queryDupNames);
    }

    // A Group key that can be passed to a separate method.
    // Override Equals and GetHashCode to define equality for the key.
    // Override ToString to provide a friendly name for Key.ToString()
    class PortableKey
    {
        public string Name { get; set; }
        public DateTime LastWriteTime { get; set; }
    }
}
```

```

public long Length { get; set; }

public override bool Equals(object obj)
{
    PortableKey other = (PortableKey)obj;
    return other.LastWriteTime == this.LastWriteTime &&
        other.Length == this.Length &&
        other.Name == this.Name;
}

public override int GetHashCode()
{
    string str = $"{this.LastWriteTime}{this.Length}{this.Name}";
    return str.GetHashCode();
}

public override string ToString()
{
    return $"{this.Name} {this.Length} {this.LastWriteTime}";
}
}

static void QueryDuplicates2()
{
    // Change the root drive or folder if necessary.
    string startFolder = @"c:\program files\Microsoft Visual Studio 9.0\Common7";

    // Make the lines shorter for the console display
    int charsToSkip = startFolder.Length;

    // Take a snapshot of the file system.
    System.IO.DirectoryInfo dir = new System.IO.DirectoryInfo(startFolder);
    IEnumerable<System.IO.FileInfo> fileList = dir.GetFiles("*.*",
System.IO.SearchOption.AllDirectories);

    // Note the use of a compound key. Files that match
    // all three properties belong to the same group.
    // A named type is used to enable the query to be
    // passed to another method. Anonymous types can also be used
    // for composite keys but cannot be passed across method boundaries
    //
    var queryDupFiles =
        from file in fileList
        group file.FullName.Substring(charsToSkip) by
            new PortableKey { Name = file.Name, LastWriteTime = file.LastWriteTime, Length = file.Length
    } into fileGroup
        where fileGroup.Count() > 1
        select fileGroup;

    var list = queryDupFiles.ToList();

    int i = queryDupFiles.Count();

    PageOutput<PortableKey, string>(queryDupFiles);
}

// A generic method to page the output of the QueryDuplications methods
// Here the type of the group must be specified explicitly. "var" cannot
// be used in method signatures. This method does not display more than one
// group per page.
private static void PageOutput<K, V>(IEnumerable<System.Linq.IGrouping<K, V>> groupByExtList)
{
    // Flag to break out of paging loop.
    bool goAgain = true;

    // "3" = 1 line for extension + 1 for "Press any key" + 1 for input cursor.
    int numLines = Console.WindowHeight - 3;

    // Iterate through the outer collection of groups.
    foreach (var filegroup in groupByExtList)
    {

```

```

// Start a new extension at the top of a page.
int currentLine = 0;

// Output only as many lines of the current group as will fit in the window.
do
{
    Console.Clear();
    Console.WriteLine("Filename = {0}", filegroup.Key.ToString() == String.Empty ? "[none]" :
filegroup.Key.ToString());

    // Get 'numLines' number of items starting at number 'currentLine'.
    var resultPage = filegroup.Skip(currentLine).Take(numLines);

    //Execute the resultPage query
    foreach (var fileName in resultPage)
    {
        Console.WriteLine("\t{0}", fileName);
    }

    // Increment the line counter.
    currentLine += numLines;

    // Give the user a chance to escape.
    Console.WriteLine("Press any key to continue or the 'End' key to break...");
    ConsoleKey key = Console.ReadKey().Key;
    if (key == ConsoleKey.End)
    {
        goAgain = false;
        break;
    }
} while (currentLine < filegroup.Count());

if (goAgain == false)
    break;
}
}
}

```

The first query uses a simple key to determine a match; this finds files that have the same name but whose contents might be different. The second query uses a compound key to match against three properties of the [FileInfo](#) object. This query is much more likely to find files that have the same name and similar or identical content.

Compiling the Code

Create a C# console application project, with `using` directives for the `System.Linq` and `System.IO` namespaces.

See also

- [LINQ to Objects \(C#\)](#)
- [LINQ and File Directories \(C#\)](#)

How to query the contents of text files in a folder (LINQ) (C#)

12/28/2021 • 2 minutes to read • [Edit Online](#)

This example shows how to query over all the files in a specified directory tree, open each file, and inspect its contents. This type of technique could be used to create indexes or reverse indexes of the contents of a directory tree. A simple string search is performed in this example. However, more complex types of pattern matching can be performed with a regular expression. For more information, see [How to combine LINQ queries with regular expressions \(C#\)](#).

Example

```

class QueryContents
{
    public static void Main()
    {
        // Modify this path as necessary.
        string startFolder = @"c:\program files\Microsoft Visual Studio 9.0\";

        // Take a snapshot of the file system.
        System.IO.DirectoryInfo dir = new System.IO.DirectoryInfo(startFolder);

        // This method assumes that the application has discovery permissions
        // for all folders under the specified path.
        IEnumerable<System.IO.FileInfo> fileList = dir.GetFiles("*.*",
System.IO.SearchOption.AllDirectories);

        string searchTerm = @"Visual Studio";

        // Search the contents of each file.
        // A regular expression created with the RegEx class
        // could be used instead of the Contains method.
        // queryMatchingFiles is an IEnumerable<string>.
        var queryMatchingFiles =
            from file in fileList
            where file.Extension == ".htm"
            let fileText = GetFileText(file.FullName)
            where fileText.Contains(searchTerm)
            select file.FullName;

        // Execute the query.
        Console.WriteLine("The term \"{0}\" was found in:", searchTerm);
        foreach (string filename in queryMatchingFiles)
        {
            Console.WriteLine(filename);
        }

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit");
        Console.ReadKey();
    }

    // Read the contents of the file.
    static string GetFileText(string name)
    {
        string fileContents = String.Empty;

        // If the file has been deleted since we took
        // the snapshot, ignore it and return the empty string.
        if (System.IO.File.Exists(name))
        {
            fileContents = System.IO.File.ReadAllText(name);
        }
        return fileContents;
    }
}

```

Compiling the Code

Create a C# console application project, with `using` directives for the System.Linq and System.IO namespaces.

See also

- [LINQ and File Directories \(C#\)](#)
- [LINQ to Objects \(C#\)](#)

How to query an ArrayList with LINQ (C#)

12/28/2021 • 2 minutes to read • [Edit Online](#)

When using LINQ to query non-generic [IEnumerable](#) collections such as [ArrayList](#), you must explicitly declare the type of the range variable to reflect the specific type of the objects in the collection. For example, if you have an [ArrayList](#) of `Student` objects, your [from clause](#) should look like this:

```
var query = from Student s in arrList
//...
```

By specifying the type of the range variable, you are casting each item in the [ArrayList](#) to a `Student`.

The use of an explicitly typed range variable in a query expression is equivalent to calling the [Cast](#) method. [Cast](#) throws an exception if the specified cast cannot be performed. [Cast](#) and [OfType](#) are the two Standard Query Operator methods that operate on non-generic [IEnumerable](#) types. For more information, see [Type Relationships in LINQ Query Operations](#).

Example

The following example shows a simple query over an [ArrayList](#). Note that this example uses object initializers when the code calls the [Add](#) method, but this is not a requirement.

```

using System;
using System.Collections;
using System.Linq;

namespace NonGenericLINQ
{
    public class Student
    {
        public string FirstName { get; set; }
        public string LastName { get; set; }
        public int[] Scores { get; set; }
    }

    class Program
    {
        static void Main(string[] args)
        {
            ArrayList arrList = new ArrayList();
            arrList.Add(
                new Student
                {
                    FirstName = "Svetlana", LastName = "Omelchenko", Scores = new int[] { 98, 92, 81, 60 }
                });
            arrList.Add(
                new Student
                {
                    FirstName = "Claire", LastName = "O'Donnell", Scores = new int[] { 75, 84, 91, 39 }
                });
            arrList.Add(
                new Student
                {
                    FirstName = "Sven", LastName = "Mortensen", Scores = new int[] { 88, 94, 65, 91 }
                });
            arrList.Add(
                new Student
                {
                    FirstName = "Cesar", LastName = "Garcia", Scores = new int[] { 97, 89, 85, 82 }
                });

            var query = from Student student in arrList
                        where student.Scores[0] > 95
                        select student;

            foreach (Student s in query)
                Console.WriteLine(s.LastName + ": " + s.Scores[0]);

            // Keep the console window open in debug mode.
            Console.WriteLine("Press any key to exit.");
            Console.ReadKey();
        }
    }
}

/* Output:
    Omelchenko: 98
    Garcia: 97
*/

```

See also

- [LINQ to Objects \(C#\)](#)

How to add custom methods for LINQ queries (C#)

12/28/2021 • 5 minutes to read • [Edit Online](#)

You extend the set of methods that you use for LINQ queries by adding extension methods to the `IEnumerable<T>` interface. For example, in addition to the standard average or maximum operations, you create a custom aggregate method to compute a single value from a sequence of values. You also create a method that works as a custom filter or a specific data transform for a sequence of values and returns a new sequence. Examples of such methods are [Distinct](#), [Skip](#), and [Reverse](#).

When you extend the `IEnumerable<T>` interface, you can apply your custom methods to any enumerable collection. For more information, see [Extension Methods](#).

Add an aggregate method

An aggregate method computes a single value from a set of values. LINQ provides several aggregate methods, including [Average](#), [Min](#), and [Max](#). You can create your own aggregate method by adding an extension method to the `IEnumerable<T>` interface.

The following code example shows how to create an extension method called `Median` to compute a median for a sequence of numbers of type `double`.

```
public static class EnumerableExtension
{
    public static double Median(this IEnumerable<double>? source)
    {
        if (source is null || !source.Any())
        {
            throw new InvalidOperationException("Cannot compute median for a null or empty set.");
        }

        var sortedList =
            source.OrderBy(number => number).ToList();

        int itemIndex = sortedList.Count / 2;

        if (sortedList.Count % 2 == 0)
        {
            // Even number of items.
            return (sortedList[itemIndex] + sortedList[itemIndex - 1]) / 2;
        }
        else
        {
            // Odd number of items.
            return sortedList[itemIndex];
        }
    }
}
```

You call this extension method for any enumerable collection in the same way you call other aggregate methods from the `IEnumerable<T>` interface.

The following code example shows how to use the `Median` method for an array of type `double`.

```
double[] numbers = { 1.9, 2, 8, 4, 5.7, 6, 7.2, 0 };
var query = numbers.Median();

Console.WriteLine($"double: Median = {query}");
// This code produces the following output:
//      double: Median = 4.85
```

Overload an aggregate method to accept various types

You can overload your aggregate method so that it accepts sequences of various types. The standard approach is to create an overload for each type. Another approach is to create an overload that will take a generic type and convert it to a specific type by using a delegate. You can also combine both approaches.

Create an overload for each type

You can create a specific overload for each type that you want to support. The following code example shows an overload of the `Median` method for the `int` type.

```
// int overload
public static double Median(this IEnumerable<int> source) =>
    (from number in source select (double)number).Median();
```

You can now call the `Median` overloads for both `integer` and `double` types, as shown in the following code:

```
double[] numbers1 = { 1.9, 2, 8, 4, 5.7, 6, 7.2, 0 };
var query1 = numbers1.Median();

Console.WriteLine($"double: Median = {query1}");

int[] numbers2 = { 1, 2, 3, 4, 5 };
var query2 = numbers2.Median();

Console.WriteLine($"int: Median = {query2}");
// This code produces the following output:
//      double: Median = 4.85
//      int: Median = 3
```

Create a generic overload

You can also create an overload that accepts a sequence of generic objects. This overload takes a delegate as a parameter and uses it to convert a sequence of objects of a generic type to a specific type.

The following code shows an overload of the `Median` method that takes the `Func<T,TResult>` delegate as a parameter. This delegate takes an object of generic type `T` and returns an object of type `double`.

```
// generic overload
public static double Median<T>(
    this IEnumerable<T> numbers, Func<T, double> selector) =>
    (from num in numbers select selector(num)).Median();
```

You can now call the `Median` method for a sequence of objects of any type. If the type doesn't have its own method overload, you have to pass a delegate parameter. In C#, you can use a lambda expression for this purpose. Also, in Visual Basic only, if you use the `Aggregate` or `Group By` clause instead of the method call, you can pass any value or expression that is in the scope of this clause.

The following example code shows how to call the `Median` method for an array of integers and an array of strings. For strings, the median for the lengths of strings in the array is calculated. The example shows how to pass the `Func<T,TResult>` delegate parameter to the `Median` method for each case.

```

int[] numbers3 = { 1, 2, 3, 4, 5 };

/*
    You can use the num => num lambda expression as a parameter for the Median method
    so that the compiler will implicitly convert its value to double.
    If there is no implicit conversion, the compiler will display an error message.
*/
var query3 = numbers3.Median(num => num);

Console.WriteLine($"int: Median = {query3}");

string[] numbers4 = { "one", "two", "three", "four", "five" };

// With the generic overload, you can also use numeric properties of objects.
var query4 = numbers4.Median(str => str.Length);

Console.WriteLine($"string: Median = {query4}");
// This code produces the following output:
//     int: Median = 3
//     string: Median = 4

```

Add a method that returns a sequence

You can extend the [IEnumerable<T>](#) interface with a custom query method that returns a sequence of values. In this case, the method must return a collection of type [IEnumerable<T>](#). Such methods can be used to apply filters or data transforms to a sequence of values.

The following example shows how to create an extension method named `AlternateElements` that returns every other element in a collection, starting from the first element.

```

// Extension method for the IEnumerable<T> interface.
// The method returns every other element of a sequence.
public static IEnumerable<T> AlternateElements<T>(this IEnumerable<T> source)
{
    int index = 0;
    foreach (T element in source)
    {
        if (index % 2 == 0)
        {
            yield return element;
        }

        index++;
    }
}

```

You can call this extension method for any enumerable collection just as you would call other methods from the [IEnumerable<T>](#) interface, as shown in the following code:

```
string[] strings = { "a", "b", "c", "d", "e" };

var query5 = strings.AlternateElements();

foreach (var element in query5)
{
    Console.WriteLine(element);
}
// This code produces the following output:
//      a
//      c
//      e
```

See also

- [IEnumerable<T>](#)
- [Extension Methods](#)

LINQ to ADO.NET (Portal Page)

12/28/2021 • 2 minutes to read • [Edit Online](#)

LINQ to ADO.NET enables you to query over any enumerable object in ADO.NET by using the Language-Integrated Query (LINQ) programming model.

NOTE

The LINQ to ADO.NET documentation is located in the ADO.NET section of the .NET Framework SDK: [LINQ and ADO.NET](#).

There are three separate ADO.NET Language-Integrated Query (LINQ) technologies: LINQ to DataSet, LINQ to SQL, and LINQ to Entities. LINQ to DataSet provides richer, optimized querying over the [DataSet](#), LINQ to SQL enables you to directly query SQL Server database schemas, and LINQ to Entities allows you to query an Entity Data Model.

LINQ to DataSet

The [DataSet](#) is one of the most widely used components in ADO.NET, and is a key element of the disconnected programming model that ADO.NET is built on. Despite this prominence, however, the [DataSet](#) has limited query capabilities.

LINQ to DataSet enables you to build richer query capabilities into [DataSet](#) by using the same query functionality that is available for many other data sources.

For more information, see [LINQ to DataSet](#).

LINQ to SQL

LINQ to SQL provides a run-time infrastructure for managing relational data as objects. In LINQ to SQL, the data model of a relational database is mapped to an object model expressed in the programming language of the developer. When you execute the application, LINQ to SQL translates language-integrated queries in the object model into SQL and sends them to the database for execution. When the database returns the results, LINQ to SQL translates them back into objects that you can manipulate.

LINQ to SQL includes support for stored procedures and user-defined functions in the database, and for inheritance in the object model.

For more information, see [LINQ to SQL](#).

LINQ to Entities

Through the Entity Data Model, relational data is exposed as objects in the .NET environment. This makes the object layer an ideal target for LINQ support, allowing developers to formulate queries against the database from the language used to build the business logic. This capability is known as LINQ to Entities. See [LINQ to Entities](#) for more information.

See also

- [LINQ and ADO.NET](#)
- [Language-Integrated Query \(LINQ\) \(C#\)](#)

Enabling a Data Source for LINQ Querying

12/28/2021 • 3 minutes to read • [Edit Online](#)

There are various ways to extend LINQ to enable any data source to be queried in the LINQ pattern. The data source might be a data structure, a Web service, a file system, or a database, to name some. The LINQ pattern makes it easy for clients to query a data source for which LINQ querying is enabled, because the syntax and pattern of the query does not change. The ways in which LINQ can be extended to these data sources include the following:

- Implementing the [IEnumerable<T>](#) interface in a type to enable LINQ to Objects querying of that type.
- Creating standard query operator methods such as [Where](#) and [Select](#) that extend a type, to enable custom LINQ querying of that type.
- Creating a provider for your data source that implements the [IQueryable<T>](#) interface. A provider that implements this interface receives LINQ queries in the form of expression trees, which it can execute in a custom way, for example remotely.
- Creating a provider for your data source that takes advantage of an existing LINQ technology. Such a provider would enable not only querying, but also insert, update, and delete operations and mapping for user-defined types.

This topic discusses these options.

How to Enable LINQ Querying of Your Data Source

In-Memory Data

There are two ways you can enable LINQ querying of in-memory data. If the data is of a type that implements [IEnumerable<T>](#), you can query the data by using LINQ to Objects. If it does not make sense to enable enumeration of your type by implementing the [IEnumerable<T>](#) interface, you can define LINQ standard query operator methods in that type or create LINQ standard query operator methods that extend the type. Custom implementations of the standard query operators should use deferred execution to return the results.

Remote Data

The best option for enabling LINQ querying of a remote data source is to implement the [IQueryable<T>](#) interface. However, this differs from extending a provider such as LINQ to SQL for a data source.

IQueryable LINQ Providers

LINQ providers that implement [IQueryable<T>](#) can vary widely in their complexity. This section discusses the different levels of complexity.

A less complex `IQueryable` provider might interface with a single method of a Web service. This type of provider is very specific because it expects specific information in the queries that it handles. It has a closed type system, perhaps exposing a single result type. Most of the execution of the query occurs locally, for example by using the [Enumerable](#) implementations of the standard query operators. A less complex provider might examine only one method call expression in the expression tree that represents the query, and let the remaining logic of the query be handled elsewhere.

An `IQueryable` provider of medium complexity might target a data source that has a partially expressive query language. If it targets a Web service, it might interface with more than one method of the Web service and select the method to call based on the question that the query poses. A provider of medium complexity would have a

richer type system than a simple provider, but it would still be a fixed type system. For example, the provider might expose types that have one-to-many relationships that can be traversed, but it would not provide mapping technology for user-defined types.

A complex `IQueryable` provider, such as the LINQ to SQL provider, might translate complete LINQ queries to an expressive query language, such as SQL. A complex provider is more general than a less complex provider, because it can handle a wider variety of questions in the query. It also has an open type system and therefore must contain extensive infrastructure to map user-defined types. Developing a complex provider requires a significant amount of effort.

See also

- [IQueryable<T>](#)
- [IEnumerable<T>](#)
- [Enumerable](#)
- [Standard Query Operators Overview \(C#\)](#)
- [LINQ to Objects \(C#\)](#)

Visual Studio IDE and Tools Support for LINQ (C#)

12/28/2021 • 2 minutes to read • [Edit Online](#)

The Visual Studio integrated development environment (IDE) provides the following features that support LINQ application development:

Object Relational Designer

The Object Relational Designer is a visual design tool that you can use in [LINQ to SQL](#) applications to generate classes in C# that represent the relational data in an underlying database. For more information, see [LINQ to SQL Tools in Visual Studio](#).

SQLMetal Command Line Tool

SQLMetal is a command-line tool that can be used in build processes to generate classes from existing databases for use in LINQ to SQL applications. For more information, see [SqlMetal.exe \(Code Generation Tool\)](#).

LINQ-Aware Code Editor

The C# code editor supports LINQ extensively with IntelliSense and formatting capabilities.

Visual Studio Debugger Support

The Visual Studio debugger supports debugging of query expressions. For more information, see [Debugging LINQ](#).

See also

- [Language-Integrated Query \(LINQ\) \(C#\)](#)

Reflection (C#)

12/28/2021 • 2 minutes to read • [Edit Online](#)

Reflection provides objects (of type [Type](#)) that describe assemblies, modules, and types. You can use reflection to dynamically create an instance of a type, bind the type to an existing object, or get the type from an existing object and invoke its methods or access its fields and properties. If you are using attributes in your code, reflection enables you to access them. For more information, see [Attributes](#).

Here's a simple example of reflection using the [GetType\(\)](#) method - inherited by all types from the `Object` base class - to obtain the type of a variable:

NOTE

Make sure you add `using System;` and `using System.Reflection;` at the top of your `.cs` file.

```
// Using GetType to obtain type information:
int i = 42;
Type type = i.GetType();
Console.WriteLine(type);
```

The output is: `System.Int32`.

The following example uses reflection to obtain the full name of the loaded assembly.

```
// Using Reflection to get information of an Assembly:
Assembly info = typeof(int).Assembly;
Console.WriteLine(info);
```

The output is: `System.Private.CoreLib, Version=4.0.0.0, Culture=neutral, PublicKeyToken=7cec85d7bea7798e`.

NOTE

The C# keywords `protected` and `internal` have no meaning in Intermediate Language (IL) and are not used in the reflection APIs. The corresponding terms in IL are *Family* and *Assembly*. To identify an `internal` method using reflection, use the `IsAssembly` property. To identify a `protected internal` method, use the `IsFamilyOrAssembly`.

Reflection overview

Reflection is useful in the following situations:

- When you have to access attributes in your program's metadata. For more information, see [Retrieving Information Stored in Attributes](#).
- For examining and instantiating types in an assembly.
- For building new types at run time. Use classes in [System.Reflection.Emit](#).
- For performing late binding, accessing methods on types created at run time. See the topic [Dynamically Loading and Using Types](#).

Related sections

For more information:

- [Reflection](#)
- [Viewing Type Information](#)
- [Reflection and Generic Types](#)
- [System.Reflection.Emit](#)
- [Retrieving Information Stored in Attributes](#)

See also

- [C# Programming Guide](#)
- [Assemblies in .NET](#)

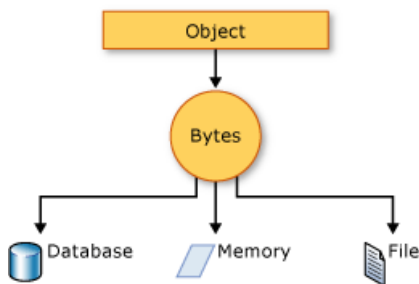
Serialization (C#)

12/28/2021 • 4 minutes to read • [Edit Online](#)

Serialization is the process of converting an object into a stream of bytes to store the object or transmit it to memory, a database, or a file. Its main purpose is to save the state of an object in order to be able to recreate it when needed. The reverse process is called deserialization.

How serialization works

This illustration shows the overall process of serialization:



The object is serialized to a stream that carries the data. The stream may also have information about the object's type, such as its version, culture, and assembly name. From that stream, the object can be stored in a database, a file, or memory.

Uses for serialization

Serialization allows the developer to save the state of an object and re-create it as needed, providing storage of objects as well as data exchange. Through serialization, a developer can perform actions such as:

- Sending the object to a remote application by using a web service
- Passing an object from one domain to another
- Passing an object through a firewall as a JSON or XML string
- Maintaining security or user-specific information across applications

JSON serialization

The [System.Text.Json](#) namespace contains classes for JavaScript Object Notation (JSON) serialization and deserialization. JSON is an open standard that is commonly used for sharing data across the web.

JSON serialization serializes the public properties of an object into a string, byte array, or stream that conforms to [the RFC 8259 JSON specification](#). To control the way [JsonSerializer](#) serializes or deserializes an instance of the class:

- Use a [JsonSerializerOptions](#) object
- Apply attributes from the [System.Text.Json.Serialization](#) namespace to classes or properties
- [Implement custom converters](#)

Binary and XML serialization

The [System.Runtime.Serialization](#) namespace contains classes for binary and XML serialization and deserialization.

Binary serialization uses binary encoding to produce compact serialization for uses such as storage or socket-

based network streams. In binary serialization, all members, even members that are read-only, are serialized, and performance is enhanced.

WARNING

Binary serialization can be dangerous. For more information, see [BinaryFormatter security guide](#).

XML serialization serializes the public fields and properties of an object, or the parameters and return values of methods, into an XML stream that conforms to a specific XML Schema definition language (XSD) document. XML serialization results in strongly typed classes with public properties and fields that are converted to XML. [System.Xml.Serialization](#) contains classes for serializing and deserializing XML. You apply attributes to classes and class members to control the way the [XmlSerializer](#) serializes or deserializes an instance of the class.

Making an object serializable

For binary or XML serialization, you need:

- The object to be serialized
- A stream to contain the serialized object
- A [System.Runtime.Serialization.Formatter](#) instance

Apply the [SerializableAttribute](#) attribute to a type to indicate that instances of the type can be serialized. An exception is thrown if you attempt to serialize but the type doesn't have the [SerializableAttribute](#) attribute.

To prevent a field from being serialized, apply the [NonSerializedAttribute](#) attribute. If a field of a serializable type contains a pointer, a handle, or some other data structure that is specific to a particular environment, and the field cannot be meaningfully reconstituted in a different environment, then you may want to make it nonserializable.

If a serialized class contains references to objects of other classes that are marked [SerializableAttribute](#), those objects will also be serialized.

Basic and custom serialization

Binary and XML serialization can be performed in two ways, basic and custom.

Basic serialization uses .NET to automatically serialize the object. The only requirement is that the class has the [SerializableAttribute](#) attribute applied. The [NonSerializedAttribute](#) can be used to keep specific fields from being serialized.

When you use basic serialization, the versioning of objects may create problems. You would use custom serialization when versioning issues are important. Basic serialization is the easiest way to perform serialization, but it does not provide much control over the process.

In custom serialization, you can specify exactly which objects will be serialized and how it will be done. The class must be marked [SerializableAttribute](#) and implement the [ISerializable](#) interface. If you want your object to be deserialized in a custom manner as well, use a custom constructor.

Designer serialization

Designer serialization is a special form of serialization that involves the kind of object persistence associated with development tools. Designer serialization is the process of converting an object graph into a source file that can later be used to recover the object graph. A source file can contain code, markup, or even SQL table information.

Related Topics and Examples

[System.Text.Json overview](#) Shows how to get the `System.Text.Json` library.

[How to serialize and deserialize JSON in .NET](#). Shows how to read and write object data to and from JSON using the `JsonSerializer` class.

[Walkthrough: Persisting an Object in Visual Studio \(C#\)](#)

Demonstrates how serialization can be used to persist an object's data between instances, allowing you to store values and retrieve them the next time the object is instantiated.

[How to read object data from an XML file \(C#\)](#)

Shows how to read object data that was previously written to an XML file using the `XmlSerializer` class.

[How to write object data to an XML file \(C#\)](#)

Shows how to write the object from a class to an XML file using the `XmlSerializer` class.

How to write object data to an XML file (C#)

12/28/2021 • 2 minutes to read • [Edit Online](#)

This example writes the object from a class to an XML file using the [XmlSerializer](#) class.

Example

```
public class XMLWrite
{
    static void Main(string[] args)
    {
        WriteXML();
    }

    public class Book
    {
        public String title;
    }

    public static void WriteXML()
    {
        Book overview = new Book();
        overview.title = "Serialization Overview";
        System.Xml.Serialization.XmlSerializer writer =
            new System.Xml.Serialization.XmlSerializer(typeof(Book));

        var path = Environment.GetFolderPath(Environment.SpecialFolder.MyDocuments) +
            "\\SerializationOverview.xml";
        System.IO.FileStream file = System.IO.File.Create(path);

        writer.Serialize(file, overview);
        file.Close();
    }
}
```

Compiling the Code

The class being serialized must have a public constructor without parameters.

Robust Programming

The following conditions may cause an exception:

- The class being serialized does not have a public, parameterless constructor.
- The file exists and is read-only ([IOException](#)).
- The path is too long ([PathTooLongException](#)).
- The disk is full ([IOException](#)).

.NET Security

This example creates a new file, if the file does not already exist. If an application needs to create a file, that application needs `Create` access for the folder. If the file already exists, the application needs only `Write` access,

a lesser privilege. Where possible, it is more secure to create the file during deployment, and only grant `Read` access to a single file, rather than `Create` access for a folder.

See also

- [StreamWriter](#)
- [How to read object data from an XML file \(C#\)](#)
- [Serialization \(C#\)](#)

How to read object data from an XML file (C#)

12/28/2021 • 2 minutes to read • [Edit Online](#)

This example reads object data that was previously written to an XML file using the [XmlSerializer](#) class.

Example

```
public class Book
{
    public String title;
}

public void ReadXML()
{
    // First write something so that there is something to read ...
    var b = new Book { title = "Serialization Overview" };
    var writer = new System.Xml.Serialization.XmlSerializer(typeof(Book));
    var wfile = new System.IO.StreamWriter(@"c:\temp\SerializationOverview.xml");
    writer.Serialize(wfile, b);
    wfile.Close();

    // Now we can read the serialized book ...
    System.Xml.Serialization.XmlSerializer reader =
        new System.Xml.Serialization.XmlSerializer(typeof(Book));
    System.IO.StreamReader file = new System.IO.StreamReader(
        @"c:\temp\SerializationOverview.xml");
    Book overview = (Book)reader.Deserialize(file);
    file.Close();

    Console.WriteLine(overview.title);
}
```

Compiling the Code

Replace the file name "c:\temp\SerializationOverview.xml" with the name of the file containing the serialized data. For more information about serializing data, see [How to write object data to an XML file \(C#\)](#).

The class must have a public constructor without parameters.

Only public properties and fields are deserialized.

Robust Programming

The following conditions may cause an exception:

- The class being serialized does not have a public, parameterless constructor.
- The data in the file does not represent data from the class to be deserialized.
- The file does not exist ([IOException](#)).

.NET Security

Always verify inputs, and never deserialize data from an untrusted source. The re-created object runs on a local computer with the permissions of the code that deserialized it. Verify all inputs before using the data in your

application.

See also

- [StreamWriter](#)
- [How to write object data to an XML file \(C#\)](#)
- [Serialization \(C#\)](#)
- [C# Programming Guide](#)

Walkthrough: persisting an object using C#

12/28/2021 • 4 minutes to read • [Edit Online](#)

You can use serialization to persist an object's data between instances, which enables you to store values and retrieve them the next time that the object is instantiated.

In this walkthrough, you will create a basic `Loan` object and persist its data to a file. You will then retrieve the data from the file when you re-create the object.

IMPORTANT

This example creates a new file if the file does not already exist. If an application must create a file, that application must have `Create` permission for the folder. Permissions are set by using access control lists. If the file already exists, the application needs only `Write` permission, a lesser permission. Where possible, it's more secure to create the file during deployment and only grant `Read` permissions to a single file (instead of Create permissions for a folder). Also, it's more secure to write data to user folders than to the root folder or the Program Files folder.

IMPORTANT

This example stores data in a binary format file. These formats should not be used for sensitive data, such as passwords or credit-card information.

Prerequisites

- To build and run, install the [.NET SDK](#).
- Install your favorite code editor, if you haven't already.

TIP

Need to install a code editor? Try [Visual Studio](#)!

- The example requires C# 7.3. See [Select the C# language version](#)

You can examine the sample code online [at the .NET samples GitHub repository](#).

Creating the loan object

The first step is to create a `Loan` class and a console application that uses the class:

1. Create a new application. Type `dotnet new console -o serialization` to create a new console application in a subdirectory named `serialization`.
2. Open the application in your editor, and add a new class named `Loan.cs`.
3. Add the following code to your `Loan` class:

```

public class Loan : INotifyPropertyChanged
{
    public double LoanAmount { get; set; }
    public double InterestRatePercent { get; set; }

    [field:NonSerialized()]
    public DateTime TimeLastLoaded { get; set; }

    public int Term { get; set; }

    private string customer;
    public string Customer
    {
        get { return customer; }
        set
        {
            customer = value;
            PropertyChanged?.Invoke(this,
                new PropertyChangedEventArgs(nameof(Customer)));
        }
    }

    [field: NonSerialized()]
    public event System.ComponentModel.PropertyChangedEventHandler PropertyChanged;

    public Loan(double loanAmount,
                double interestRate,
                int term,
                string customer)
    {
        this.LoanAmount = loanAmount;
        this.InterestRatePercent = interestRate;
        this.Term = term;
        this.customer = customer;
    }
}

```

You will also have to create an application that uses the `Loan` class.

Serialize the loan object

1. Open `Program.cs`. Add the following code:

```

Loan TestLoan = new Loan(10000.0, 7.5, 36, "Neil Black");

```

Add an event handler for the `PropertyChanged` event, and a few lines to modify the `Loan` object and display the changes. You can see the additions in the following code:

```

TestLoan.PropertyChanged += (_, __) => Console.WriteLine($"New customer value: {TestLoan.Customer}");

TestLoan.Customer = "Henry Clay";
Console.WriteLine(TestLoan.InterestRatePercent);
TestLoan.InterestRatePercent = 7.1;
Console.WriteLine(TestLoan.InterestRatePercent);

```

At this point, you can run the code, and see the current output:

```
New customer value: Henry Clay
7.5
7.1
```

Running this application repeatedly always writes the same values. A new `Loan` object is created every time you run the program. In the real world, interest rates change periodically, but not necessarily every time that the application is run. Serialization code means you preserve the most recent interest rate between instances of the application. In the next step, you will do just that by adding serialization to the `Loan` class.

Using Serialization to Persist the Object

In order to persist the values for the `Loan` class, you must first mark the class with the `Serializable` attribute. Add the following code above the `Loan` class definition:

```
[Serializable()]
```

The `SerializableAttribute` tells the compiler that everything in the class can be persisted to a file. Because the `PropertyChanged` event does not represent part of the object graph that should be stored, it should not be serialized. Doing so would serialize all objects that are attached to that event. You can add the `NonSerializedAttribute` to the field declaration for the `PropertyChanged` event handler.

```
[field: NonSerialized()]
public event System.ComponentModel.PropertyChangedEventHandler PropertyChanged;
```

Beginning with C# 7.3, you can attach attributes to the backing field of an auto-implemented property using the `field` target value. The following code adds a `TimeLastLoaded` property and marks it as not serializable:

```
[field:NonSerialized()]
public DateTime TimeLastLoaded { get; set; }
```

The next step is to add the serialization code to the `LoanApp` application. In order to serialize the class and write it to a file, you use the `System.IO` and `System.Runtime.Serialization.Formatters.Binary` namespaces. To avoid typing the fully qualified names, you can add references to the necessary namespaces as shown in the following code:

```
using System.IO;
using System.Runtime.Serialization.Formatters.Binary;
```

The next step is to add code to deserialize the object from the file when the object is created. Add a constant to the class for the serialized data's file name as shown in the following code:

```
const string FileName = @"../../SavedLoan.bin";
```

Next, add the following code after the line that creates the `TestLoan` object:

```
if (File.Exists(FileName))
{
    Console.WriteLine("Reading saved file");
    Stream openFileStream = File.OpenRead(FileName);
    BinaryFormatter deserializer = new BinaryFormatter();
    TestLoan = (Loan)deserializer.Deserialize(openFileStream);
    TestLoan.TimeLastLoaded = DateTime.Now;
    openFileStream.Close();
}
```

You first must check that the file exists. If it exists, create a [Stream](#) class to read the binary file and a [BinaryFormatter](#) class to translate the file. You also need to convert from the stream type to the Loan object type.

Next you must add code to serialize the class to a file. Add the following code after the existing code in the Main method:

```
Stream SaveFileStream = File.Create(FileName);
BinaryFormatter serializer = new BinaryFormatter();
serializer.Serialize(SaveFileStream, TestLoan);
SaveFileStream.Close();
```

At this point, you can again build and run the application. The first time it runs, notice that the interest rates starts at 7.5, and then changes to 7.1. Close the application and then run it again. Now, the application prints the message that it has read the saved file, and the interest rate is 7.1 even before the code that changes it.

See also

- [Serialization \(C#\)](#)
- [C# Programming Guide](#)

Statements, Expressions, and Operators (C# Programming Guide)

12/28/2021 • 2 minutes to read • [Edit Online](#)

The C# code that comprises an application consists of statements made up of keywords, expressions and operators. This section contains information regarding these fundamental elements of a C# program.

For more information, see:

- [Statements](#)
- [Operators and expressions](#)
- [Expression-bodied members](#)
- [Equality Comparisons](#)

C# Language Specification

For more information, see the [C# Language Specification](#). The language specification is the definitive source for C# syntax and usage.

See also

- [C# Programming Guide](#)
- [Casting and Type Conversions](#)

Statements (C# Programming Guide)

12/28/2021 • 6 minutes to read • [Edit Online](#)

The actions that a program takes are expressed in statements. Common actions include declaring variables, assigning values, calling methods, looping through collections, and branching to one or another block of code, depending on a given condition. The order in which statements are executed in a program is called the flow of control or flow of execution. The flow of control may vary every time that a program is run, depending on how the program reacts to input that it receives at run time.

A statement can consist of a single line of code that ends in a semicolon, or a series of single-line statements in a block. A statement block is enclosed in {} brackets and can contain nested blocks. The following code shows two examples of single-line statements, and a multi-line statement block:

```
static void Main()
{
    // Declaration statement.
    int counter;

    // Assignment statement.
    counter = 1;

    // Error! This is an expression, not an expression statement.
    // counter + 1;

    // Declaration statements with initializers are functionally
    // equivalent to declaration statement followed by assignment statement:
    int[] radii = { 15, 32, 108, 74, 9 }; // Declare and initialize an array.
    const double pi = 3.14159; // Declare and initialize constant.

    // foreach statement block that contains multiple statements.
    foreach (int radius in radii)
    {
        // Declaration statement with initializer.
        double circumference = pi * (2 * radius);

        // Expression statement (method invocation). A single-line
        // statement can span multiple text lines because line breaks
        // are treated as white space, which is ignored by the compiler.
        System.Console.WriteLine("Radius of circle #{0} is {1}. Circumference = {2:N2}",
                                counter, radius, circumference);

        // Expression statement (postfix increment).
        counter++;
    } // End of foreach statement block
} // End of Main method body.
} // End of SimpleStatements class.
/*
Output:
Radius of circle #1 = 15. Circumference = 94.25
Radius of circle #2 = 32. Circumference = 201.06
Radius of circle #3 = 108. Circumference = 678.58
Radius of circle #4 = 74. Circumference = 464.96
Radius of circle #5 = 9. Circumference = 56.55
*/
```

Types of statements

The following table lists the various types of statements in C# and their associated keywords, with links to topics

that include more information:

CATEGORY	C# KEYWORDS / NOTES
Declaration statements	A declaration statement introduces a new variable or constant. A variable declaration can optionally assign a value to the variable. In a constant declaration, the assignment is required.
Expression statements	Expression statements that calculate a value must store the value in a variable.
Selection statements	Selection statements enable you to branch to different sections of code, depending on one or more specified conditions. For more information, see the following topics: <ul style="list-style-type: none">• if• switch
Iteration statements	Iteration statements enable you to loop through collections like arrays, or perform the same set of statements repeatedly until a specified condition is met. For more information, see the following topics: <ul style="list-style-type: none">• do• for• foreach• while
Jump statements	Jump statements transfer control to another section of code. For more information, see the following topics: <ul style="list-style-type: none">• break• continue• goto• return• yield
Exception handling statements	Exception handling statements enable you to gracefully recover from exceptional conditions that occur at run time. For more information, see the following topics: <ul style="list-style-type: none">• throw• try-catch• try-finally• try-catch-finally
Checked and unchecked	Checked and unchecked statements enable you to specify whether numerical operations are allowed to cause an overflow when the result is stored in a variable that is too small to hold the resulting value. For more information, see checked and unchecked .

CATEGORY	C# KEYWORDS / NOTES
The <code>await</code> statement	<p>If you mark a method with the async modifier, you can use the await operator in the method. When control reaches an <code>await</code> expression in the async method, control returns to the caller, and progress in the method is suspended until the awaited task completes. When the task is complete, execution can resume in the method.</p> <p>For a simple example, see the "Async Methods" section of Methods. For more information, see Asynchronous Programming with async and await.</p>
The <code>yield return</code> statement	<p>An iterator performs a custom iteration over a collection, such as a list or an array. An iterator uses the yield return statement to return each element one at a time. When a <code>yield return</code> statement is reached, the current location in code is remembered. Execution is restarted from that location when the iterator is called the next time.</p> <p>For more information, see Iterators.</p>
The <code>fixed</code> statement	<p>The fixed statement prevents the garbage collector from relocating a movable variable. For more information, see fixed.</p>
The <code>lock</code> statement	<p>The lock statement enables you to limit access to blocks of code to only one thread at a time. For more information, see lock.</p>
Labeled statements	<p>You can give a statement a label and then use the goto keyword to jump to the labeled statement. (See the example in the following row.)</p>
The empty statement	<p>The empty statement consists of a single semicolon. It does nothing and can be used in places where a statement is required but no action needs to be performed.</p>

Declaration statements

The following code shows examples of variable declarations with and without an initial assignment, and a constant declaration with the necessary initialization.

```
// Variable declaration statements.
double area;
double radius = 2;

// Constant declaration statement.
const double pi = 3.14159;
```

Expression statements

The following code shows examples of expression statements, including assignment, object creation with assignment, and method invocation.

```
// Expression statement (assignment).
area = 3.14 * (radius * radius);

// Error. Not statement because no assignment:
//circ * 2;

// Expression statement (method invocation).
System.Console.WriteLine();

// Expression statement (new object creation).
System.Collections.Generic.List<string> strings =
    new System.Collections.Generic.List<string>();
```

The empty statement

The following examples show two uses for an empty statement:

```
void ProcessMessages()
{
    while (ProcessMessage())
        ; // Statement needed here.
}

void F()
{
    //...
    if (done) goto exit;
    //...
exit:
    ; // Statement needed here.
}
```

Embedded statements

Some statements, for example, [iteration statements](#), always have an embedded statement that follows them. This embedded statement may be either a single statement or multiple statements enclosed by {} brackets in a statement block. Even single-line embedded statements can be enclosed in {} brackets, as shown in the following example:

```
// Recommended style. Embedded statement in block.
foreach (string s in System.IO.Directory.GetDirectories(
    System.Environment.CurrentDirectory))
{
    System.Console.WriteLine(s);
}

// Not recommended.
foreach (string s in System.IO.Directory.GetDirectories(
    System.Environment.CurrentDirectory))
    System.Console.WriteLine(s);
```

An embedded statement that is not enclosed in {} brackets cannot be a declaration statement or a labeled statement. This is shown in the following example:

```
if(pointB == true)
    //Error CS1023:
    int radius = 5;
```

Put the embedded statement in a block to fix the error:

```
if (b == true)
{
    // OK:
    System.DateTime d = System.DateTime.Now;
    System.Console.WriteLine(d.ToLongDateString());
}
```

Nested statement blocks

Statement blocks can be nested, as shown in the following code:

```
foreach (string s in System.IO.Directory.GetDirectories(
    System.Environment.CurrentDirectory))
{
    if (s.StartsWith("CSharp"))
    {
        if (s.EndsWith("TempFolder"))
        {
            return s;
        }
    }
}
return "Not found.";
```

Unreachable statements

If the compiler determines that the flow of control can never reach a particular statement under any circumstances, it will produce warning CS0162, as shown in the following example:

```
// An over-simplified example of unreachable code.
const int val = 5;
if (val < 4)
{
    System.Console.WriteLine("I'll never write anything."); //CS0162
}
```

C# language specification

For more information, see the [Statements](#) section of the [C# language specification](#).

See also

- [C# Programming Guide](#)
- [Statement keywords](#)
- [C# operators and expressions](#)

Expression-bodied members (C# programming guide)

12/28/2021 • 3 minutes to read • [Edit Online](#)

Expression body definitions let you provide a member's implementation in a very concise, readable form. You can use an expression body definition whenever the logic for any supported member, such as a method or property, consists of a single expression. An expression body definition has the following general syntax:

```
member => expression;
```

where *expression* is a valid expression.

Support for expression body definitions was introduced for methods and read-only properties in C# 6 and was expanded in C# 7.0. Expression body definitions can be used with the type members listed in the following table:

MEMBER	SUPPORTED AS OF...
Method	C# 6
Read-only property	C# 6
Property	C# 7.0
Constructor	C# 7.0
Finalizer	C# 7.0
Indexer	C# 7.0

Methods

An expression-bodied method consists of a single expression that returns a value whose type matches the method's return type, or, for methods that return `void`, that performs some operation. For example, types that override the [ToString](#) method typically include a single expression that returns the string representation of the current object.

The following example defines a `Person` class that overrides the [ToString](#) method with an expression body definition. It also defines a `DisplayName` method that displays a name to the console. Note that the `return` keyword is not used in the `ToString` expression body definition.

```

using System;

public class Person
{
    public Person(string firstName, string lastName)
    {
        fname = firstName;
        lname = lastName;
    }

    private string fname;
    private string lname;

    public override string ToString() => $"{fname} {lname}".Trim();
    public void DisplayName() => Console.WriteLine(ToString());
}

class Example
{
    static void Main()
    {
        Person p = new Person("Mandy", "Dejesus");
        Console.WriteLine(p);
        p.DisplayName();
    }
}

```

For more information, see [Methods \(C# Programming Guide\)](#).

Read-only properties

Starting with C# 6, you can use expression body definition to implement a read-only property. To do that, use the following syntax:

```
PropertyType PropertyName => expression;
```

The following example defines a `Location` class whose read-only `Name` property is implemented as an expression body definition that returns the value of the private `locationName` field:

```

public class Location
{
    private string locationName;

    public Location(string name)
    {
        locationName = name;
    }

    public string Name => locationName;
}

```

For more information about properties, see [Properties \(C# Programming Guide\)](#).

Properties

Starting with C# 7.0, you can use expression body definitions to implement property `get` and `set` accessors. The following example demonstrates how to do that:

```
public class Location
{
    private string locationName;

    public Location(string name) => Name = name;

    public string Name
    {
        get => locationName;
        set => locationName = value;
    }
}
```

For more information about properties, see [Properties \(C# Programming Guide\)](#).

Constructors

An expression body definition for a constructor typically consists of a single assignment expression or a method call that handles the constructor's arguments or initializes instance state.

The following example defines a `Location` class whose constructor has a single string parameter named *name*. The expression body definition assigns the argument to the `Name` property.

```
public class Location
{
    private string locationName;

    public Location(string name) => Name = name;

    public string Name
    {
        get => locationName;
        set => locationName = value;
    }
}
```

For more information, see [Constructors \(C# Programming Guide\)](#).

Finalizers

An expression body definition for a finalizer typically contains cleanup statements, such as statements that release unmanaged resources.

The following example defines a finalizer that uses an expression body definition to indicate that the finalizer has been called.

```
using System;

public class Destroyer
{
    public override string ToString() => GetType().Name;

    ~Destroyer() => Console.WriteLine($"The {ToString()} finalizer is executing.");
}
```

For more information, see [Finalizers \(C# Programming Guide\)](#).

Indexers

Like with properties, indexer `get` and `set` accessors consist of expression body definitions if the `get` accessor consists of a single expression that returns a value or the `set` accessor performs a simple assignment.

The following example defines a class named `Sports` that includes an internal `String` array that contains the names of a number of sports. Both the indexer `get` and `set` accessors are implemented as expression body definitions.

```
using System;
using System.Collections.Generic;

public class Sports
{
    private string[] types = { "Baseball", "Basketball", "Football",
                               "Hockey", "Soccer", "Tennis",
                               "Volleyball" };

    public string this[int i]
    {
        get => types[i];
        set => types[i] = value;
    }
}
```

For more information, see [Indexers \(C# Programming Guide\)](#).

Equality comparisons (C# Programming Guide)

12/28/2021 • 3 minutes to read • [Edit Online](#)

It is sometimes necessary to compare two values for equality. In some cases, you are testing for *value equality*, also known as *equivalence*, which means that the values that are contained by the two variables are equal. In other cases, you have to determine whether two variables refer to the same underlying object in memory. This type of equality is called *reference equality*, or *identity*. This topic describes these two kinds of equality and provides links to other topics for more information.

Reference equality

Reference equality means that two object references refer to the same underlying object. This can occur through simple assignment, as shown in the following example.

```
using System;
class Test
{
    public int Num { get; set; }
    public string Str { get; set; }

    static void Main()
    {
        Test a = new Test() { Num = 1, Str = "Hi" };
        Test b = new Test() { Num = 1, Str = "Hi" };

        bool areEqual = System.Object.ReferenceEquals(a, b);
        // False:
        System.Console.WriteLine("ReferenceEquals(a, b) = {0}", areEqual);

        // Assign b to a.
        b = a;

        // Repeat calls with different results.
        areEqual = System.Object.ReferenceEquals(a, b);
        // True:
        System.Console.WriteLine("ReferenceEquals(a, b) = {0}", areEqual);

        // Keep the console open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
```

In this code, two objects are created, but after the assignment statement, both references refer to the same object. Therefore they have reference equality. Use the [ReferenceEquals](#) method to determine whether two references refer to the same object.

The concept of reference equality applies only to reference types. Value type objects cannot have reference equality because when an instance of a value type is assigned to a variable, a copy of the value is made. Therefore you can never have two unboxed structs that refer to the same location in memory. Furthermore, if you use [ReferenceEquals](#) to compare two value types, the result will always be `false`, even if the values that are contained in the objects are all identical. This is because each variable is boxed into a separate object instance. For more information, see [How to test for reference equality \(Identity\)](#).

Value equality

Value equality means that two objects contain the same value or values. For primitive value types such as [int](#) or [bool](#), tests for value equality are straightforward. You can use the `==` operator, as shown in the following example.

```
int a = GetOriginalValue();
int b = GetCurrentValue();

// Test for value equality.
if (b == a)
{
    // The two integers are equal.
}
```

For most other types, testing for value equality is more complex because it requires that you understand how the type defines it. For classes and structs that have multiple fields or properties, value equality is often defined to mean that all fields or properties have the same value. For example, two `Point` objects might be defined to be equivalent if `pointA.X` is equal to `pointB.X` and `pointA.Y` is equal to `pointB.Y`. For records, value equality means that two variables of a record type are equal if the types match and all property and field values match.

However, there is no requirement that equivalence be based on all the fields in a type. It can be based on a subset. When you compare types that you do not own, you should make sure to understand specifically how equivalence is defined for that type. For more information about how to define value equality in your own classes and structs, see [How to define value equality for a type](#).

Value equality for floating-point values

Equality comparisons of floating-point values ([double](#) and [float](#)) are problematic because of the imprecision of floating-point arithmetic on binary computers. For more information, see the remarks in the topic [System.Double](#).

Related topics

TITLE	DESCRIPTION
How to test for reference equality (Identity)	Describes how to determine whether two variables have reference equality.
How to define value equality for a type	Describes how to provide a custom definition of value equality for a type.
C# Programming Guide	Provides links to detailed information about important C# language features and features that are available to C# through .NET.
Types	Provides information about the C# type system and links to additional information.
Records	Provides information about record types, which test for value equality by default.

See also

- [C# Programming Guide](#)

How to define value equality for a class or struct (C# Programming Guide)

12/28/2021 • 9 minutes to read • [Edit Online](#)

[Records](#) automatically implement value equality. Consider defining a `record` instead of a `class` when your type models data and should implement value equality.

When you define a class or struct, you decide whether it makes sense to create a custom definition of value equality (or equivalence) for the type. Typically, you implement value equality when you expect to add objects of the type to a collection, or when their primary purpose is to store a set of fields or properties. You can base your definition of value equality on a comparison of all the fields and properties in the type, or you can base the definition on a subset.

In either case, and in both classes and structs, your implementation should follow the five guarantees of equivalence (for the following rules, assume that `x`, `y` and `z` are not null):

1. The reflexive property: `x.Equals(x)` returns `true`.
2. The symmetric property: `x.Equals(y)` returns the same value as `y.Equals(x)`.
3. The transitive property: if `(x.Equals(y) && y.Equals(z))` returns `true`, then `x.Equals(z)` returns `true`.
4. Successive invocations of `x.Equals(y)` return the same value as long as the objects referenced by `x` and `y` aren't modified.
5. Any non-null value isn't equal to null. However, `x.Equals(y)` throws an exception when `x` is null. That breaks rules 1 or 2, depending on the argument to `Equals`.

Any struct that you define already has a default implementation of value equality that it inherits from the [System.ValueType](#) override of the [Object.Equals\(Object\)](#) method. This implementation uses reflection to examine all the fields and properties in the type. Although this implementation produces correct results, it is relatively slow compared to a custom implementation that you write specifically for the type.

The implementation details for value equality are different for classes and structs. However, both classes and structs require the same basic steps for implementing equality:

1. Override the [virtual Object.Equals\(Object\)](#) method. In most cases, your implementation of `bool Equals(object obj)` should just call into the type-specific `Equals` method that is the implementation of the [System.IEquatable<T>](#) interface. (See step 2.)
2. Implement the [System.IEquatable<T>](#) interface by providing a type-specific `Equals` method. This is where the actual equivalence comparison is performed. For example, you might decide to define equality by comparing only one or two fields in your type. Don't throw exceptions from `Equals`. For classes that are related by inheritance:
 - This method should examine only fields that are declared in the class. It should call `base.Equals` to examine fields that are in the base class. (Don't call `base.Equals` if the type inherits directly from [Object](#), because the [Object](#) implementation of [Object.Equals\(Object\)](#) performs a reference equality check.)
 - Two variables should be deemed equal only if the run-time types of the variables being compared are the same. Also, make sure that the [IEquatable](#) implementation of the `Equals` method for the run-time type is used if the run-time and compile-time types of a variable are different. One

strategy for making sure run-time types are always compared correctly is to implement `IEquatable` only in `sealed` classes. For more information, see the [class example](#) later in this article.

3. Optional but recommended: Overload the `==` and `!=` operators.
4. Override `Object.GetHashCode` so that two objects that have value equality produce the same hash code.
5. Optional: To support definitions for "greater than" or "less than," implement the `IComparable<T>` interface for your type, and also overload the `<=` and `>=` operators.

NOTE

Starting in C# 9.0, you can use records to get value equality semantics without any unnecessary boilerplate code.

Class example

The following example shows how to implement value equality in a class (reference type).

```
using System;

namespace ValueEqualityClass
{
    class TwoDPoint : IEquatable<TwoDPoint>
    {
        public int X { get; private set; }
        public int Y { get; private set; }

        public TwoDPoint(int x, int y)
        {
            if (x is (< 1 or > 2000) || y is (< 1 or > 2000))
            {
                throw new ArgumentException("Point must be in range 1 - 2000");
            }
            this.X = x;
            this.Y = y;
        }

        public override bool Equals(object obj) => this.Equals(obj as TwoDPoint);

        public bool Equals(TwoDPoint p)
        {
            if (p is null)
            {
                return false;
            }

            // Optimization for a common success case.
            if (Object.ReferenceEquals(this, p))
            {
                return true;
            }

            // If run-time types are not exactly the same, return false.
            if (this.GetType() != p.GetType())
            {
                return false;
            }

            // Return true if the fields match.
            // Note that the base class is not invoked because it is
            // System.Object, which defines Equals as reference equality.
            return (X == p.X) && (Y == p.Y);
        }
    }
}
```

```

    public override int GetHashCode() => (X, Y).GetHashCode();

    public static bool operator ==(TwoDPoint lhs, TwoDPoint rhs)
    {
        if (lhs is null)
        {
            if (rhs is null)
            {
                return true;
            }

            // Only the left side is null.
            return false;
        }
        // Equals handles case of null on right side.
        return lhs.Equals(rhs);
    }

    public static bool operator !=(TwoDPoint lhs, TwoDPoint rhs) => !(lhs == rhs);
}

// For the sake of simplicity, assume a ThreeDPoint IS a TwoDPoint.
class ThreeDPoint : TwoDPoint, IEquatable<ThreeDPoint>
{
    public int Z { get; private set; }

    public ThreeDPoint(int x, int y, int z)
        : base(x, y)
    {
        if ((z < 1) || (z > 2000))
        {
            throw new ArgumentException("Point must be in range 1 - 2000");
        }
        this.Z = z;
    }

    public override bool Equals(object obj) => this.Equals(obj as ThreeDPoint);

    public bool Equals(ThreeDPoint p)
    {
        if (p is null)
        {
            return false;
        }

        // Optimization for a common success case.
        if (Object.ReferenceEquals(this, p))
        {
            return true;
        }

        // Check properties that this class declares.
        if (Z == p.Z)
        {
            // Let base class check its own fields
            // and do the run-time type comparison.
            return base.Equals((TwoDPoint)p);
        }
        else
        {
            return false;
        }
    }

    public override int GetHashCode() => (X, Y, Z).GetHashCode();

    public static bool operator ==(ThreeDPoint lhs, ThreeDPoint rhs)
    {

```

```

    {
        if (lhs is null)
        {
            if (rhs is null)
            {
                // null == null = true.
                return true;
            }

            // Only the left side is null.
            return false;
        }
        // Equals handles the case of null on right side.
        return lhs.Equals(rhs);
    }

    public static bool operator !=(ThreeDPoint lhs, ThreeDPoint rhs) => !(lhs == rhs);
}

class Program
{
    static void Main(string[] args)
    {
        ThreeDPoint pointA = new ThreeDPoint(3, 4, 5);
        ThreeDPoint pointB = new ThreeDPoint(3, 4, 5);
        ThreeDPoint pointC = null;
        int i = 5;

        Console.WriteLine("pointA.Equals(pointB) = {0}", pointA.Equals(pointB));
        Console.WriteLine("pointA == pointB = {0}", pointA == pointB);
        Console.WriteLine("null comparison = {0}", pointA.Equals(pointC));
        Console.WriteLine("Compare to some other type = {0}", pointA.Equals(i));

        TwoDPoint pointD = null;
        TwoDPoint pointE = null;

        Console.WriteLine("Two null TwoDPoints are equal: {0}", pointD == pointE);

        pointE = new TwoDPoint(3, 4);
        Console.WriteLine("(pointE == pointA) = {0}", pointE == pointA);
        Console.WriteLine("(pointA == pointE) = {0}", pointA == pointE);
        Console.WriteLine("(pointA != pointE) = {0}", pointA != pointE);

        System.Collections.ArrayList list = new System.Collections.ArrayList();
        list.Add(new ThreeDPoint(3, 4, 5));
        Console.WriteLine("pointE.Equals(list[0]): {0}", pointE.Equals(list[0]));

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}

/* Output:
pointA.Equals(pointB) = True
pointA == pointB = True
null comparison = False
Compare to some other type = False
Two null TwoDPoints are equal: True
(pointE == pointA) = False
(pointA == pointE) = False
(pointA != pointE) = True
pointE.Equals(list[0]): False
*/
}

```

On classes (reference types), the default implementation of both [Object.Equals\(Object\)](#) methods performs a reference equality comparison, not a value equality check. When an implementer overrides the virtual method,

the purpose is to give it value equality semantics.

The `==` and `!=` operators can be used with classes even if the class does not overload them. However, the default behavior is to perform a reference equality check. In a class, if you overload the `Equals` method, you should overload the `==` and `!=` operators, but it is not required.

IMPORTANT

The preceding example code may not handle every inheritance scenario the way you expect. Consider the following code:

```
TwoDPoint p1 = new ThreeDPoint(1, 2, 3);
TwoDPoint p2 = new ThreeDPoint(1, 2, 4);
Console.WriteLine(p1.Equals(p2)); // output: True
```

This code reports that `p1` equals `p2` despite the difference in `z` values. The difference is ignored because the compiler picks the `TwoDPoint` implementation of `IEquatable` based on the compile-time type.

The built-in value equality of `record` types handles scenarios like this correctly. If `TwoDPoint` and `ThreeDPoint` were `record` types, the result of `p1.Equals(p2)` would be `False`. For more information, see [Equality in record type inheritance hierarchies](#).

Struct example

The following example shows how to implement value equality in a struct (value type):

```
using System;

namespace ValueEqualityStruct
{
    struct TwoDPoint : IEquatable<TwoDPoint>
    {
        public int X { get; private set; }
        public int Y { get; private set; }

        public TwoDPoint(int x, int y)
            : this()
        {
            if (x is (< 1 or > 2000) || y is (< 1 or > 2000))
            {
                throw new ArgumentException("Point must be in range 1 - 2000");
            }
            X = x;
            Y = y;
        }

        public override bool Equals(object obj) => obj is TwoDPoint other && this.Equals(other);

        public bool Equals(TwoDPoint p) => X == p.X && Y == p.Y;

        public override int GetHashCode() => (X, Y).GetHashCode();

        public static bool operator ==(TwoDPoint lhs, TwoDPoint rhs) => lhs.Equals(rhs);

        public static bool operator !=(TwoDPoint lhs, TwoDPoint rhs) => !(lhs == rhs);
    }

    class Program
    {
        static void Main(string[] args)
        {
            TwoDPoint pointA = new TwoDPoint(3, 4);
            TwoDPoint pointB = new TwoDPoint(3, 4);
            Console.WriteLine(pointA == pointB); // output: True
        }
    }
}
```

```

    int i = 5;

    // True:
    Console.WriteLine("pointA.Equals(pointB) = {0}", pointA.Equals(pointB));
    // True:
    Console.WriteLine("pointA == pointB = {0}", pointA == pointB);
    // True:
    Console.WriteLine("object.Equals(pointA, pointB) = {0}", object.Equals(pointA, pointB));
    // False:
    Console.WriteLine("pointA.Equals(null) = {0}", pointA.Equals(null));
    // False:
    Console.WriteLine("(pointA == null) = {0}", pointA == null);
    // True:
    Console.WriteLine("(pointA != null) = {0}", pointA != null);
    // False:
    Console.WriteLine("pointA.Equals(i) = {0}", pointA.Equals(i));
    // CS0019:
    // Console.WriteLine("pointA == i = {0}", pointA == i);

    // Compare unboxed to boxed.
    System.Collections.ArrayList list = new System.Collections.ArrayList();
    list.Add(new TwoDPoint(3, 4));
    // True:
    Console.WriteLine("pointA.Equals(list[0]): {0}", pointA.Equals(list[0]));

    // Compare nullable to nullable and to non-nullable.
    TwoDPoint? pointC = null;
    TwoDPoint? pointD = null;
    // False:
    Console.WriteLine("pointA == (pointC = null) = {0}", pointA == pointC);
    // True:
    Console.WriteLine("pointC == pointD = {0}", pointC == pointD);

    TwoDPoint temp = new TwoDPoint(3, 4);
    pointC = temp;
    // True:
    Console.WriteLine("pointA == (pointC = 3,4) = {0}", pointA == pointC);

    pointD = temp;
    // True:
    Console.WriteLine("pointD == (pointC = 3,4) = {0}", pointD == pointC);

    Console.WriteLine("Press any key to exit.");
    Console.ReadKey();
}
}

/* Output:
pointA.Equals(pointB) = True
pointA == pointB = True
Object.Equals(pointA, pointB) = True
pointA.Equals(null) = False
(pointA == null) = False
(pointA != null) = True
pointA.Equals(i) = False
pointA.Equals(list[0]): True
pointA == (pointC = null) = False
pointC == pointD = True
pointA == (pointC = 3,4) = True
pointD == (pointC = 3,4) = True
*/
}

```

For structs, the default implementation of [Object.Equals\(Object\)](#) (which is the overridden version in [System.ValueType](#)) performs a value equality check by using reflection to compare the values of every field in the type. When an implementer overrides the virtual `Equals` method in a struct, the purpose is to provide a more efficient means of performing the value equality check and optionally to base the comparison on some

subset of the struct's field or properties.

The `==` and `!=` operators can't operate on a struct unless the struct explicitly overloads them.

See also

- [Equality comparisons](#)
- [C# programming guide](#)

How to test for reference equality (Identity) (C# Programming Guide)

12/28/2021 • 3 minutes to read • [Edit Online](#)

You do not have to implement any custom logic to support reference equality comparisons in your types. This functionality is provided for all types by the static [Object.ReferenceEquals](#) method.

The following example shows how to determine whether two variables have *reference equality*, which means that they refer to the same object in memory.

The example also shows why [Object.ReferenceEquals](#) always returns `false` for value types and why you should not use [ReferenceEquals](#) to determine string equality.

Example

```
using System;
using System.Text;

namespace TestReferenceEquality
{
    struct TestStruct
    {
        public int Num { get; private set; }
        public string Name { get; private set; }

        public TestStruct(int i, string s) : this()
        {
            Num = i;
            Name = s;
        }
    }

    class TestClass
    {
        public int Num { get; set; }
        public string Name { get; set; }
    }

    class Program
    {
        static void Main()
        {
            // Demonstrate reference equality with reference types.
            #region ReferenceTypes

            // Create two reference type instances that have identical values.
            TestClass tcA = new TestClass() { Num = 1, Name = "New TestClass" };
            TestClass tcB = new TestClass() { Num = 1, Name = "New TestClass" };

            Console.WriteLine("ReferenceEquals(tcA, tcB) = {0}",
                Object.ReferenceEquals(tcA, tcB)); // false

            // After assignment, tcB and tcA refer to the same object.
            // They now have reference equality.
            tcB = tcA;
            Console.WriteLine("After assignment: ReferenceEquals(tcA, tcB) = {0}",
                Object.ReferenceEquals(tcA, tcB)); // true

            // Changes made to tcA are reflected in tcB. Therefore, objects
```

```

// that have reference equality also have value equality.
tcA.Num = 42;
tcA.Name = "TestClass 42";
Console.WriteLine("tcB.Name = {0} tcB.Num: {1}", tcB.Name, tcB.Num);
#endregion

// Demonstrate that two value type instances never have reference equality.
#region ValueTypes

TestStruct tsC = new TestStruct( 1, "TestStruct 1");

// Value types are copied on assignment. tsD and tsC have
// the same values but are not the same object.
TestStruct tsD = tsC;
Console.WriteLine("After assignment: ReferenceEquals(tsC, tsD) = {0}",
    Object.ReferenceEquals(tsC, tsD)); // false

#endregion

#region stringRefEquality
// Constant strings within the same assembly are always interned by the runtime.
// This means they are stored in the same location in memory. Therefore,
// the two strings have reference equality although no assignment takes place.
string strA = "Hello world!";
string strB = "Hello world!";
Console.WriteLine("ReferenceEquals(strA, strB) = {0}",
    Object.ReferenceEquals(strA, strB)); // true

// After a new string is assigned to strA, strA and strB
// are no longer interned and no longer have reference equality.
strA = "Goodbye world!";
Console.WriteLine("strA = \"{0}\" strB = \"{1}\"", strA, strB);

Console.WriteLine("After strA changes, ReferenceEquals(strA, strB) = {0}",
    Object.ReferenceEquals(strA, strB)); // false

// A string that is created at runtime cannot be interned.
StringBuilder sb = new StringBuilder("Hello world!");
string stringC = sb.ToString();
// False:
Console.WriteLine("ReferenceEquals(stringC, strB) = {0}",
    Object.ReferenceEquals(stringC, strB));

// The string class overloads the == operator to perform an equality comparison.
Console.WriteLine("stringC == strB = {0}", stringC == strB); // true

#endregion

// Keep the console open in debug mode.
Console.WriteLine("Press any key to exit.");
Console.ReadKey();
}
}

/* Output:
ReferenceEquals(tcA, tcB) = False
After assignment: ReferenceEquals(tcA, tcB) = True
tcB.Name = TestClass 42 tcB.Num: 42
After assignment: ReferenceEquals(tsC, tsD) = False
ReferenceEquals(strA, strB) = True
strA = "Goodbye world!" strB = "Hello world!"
After strA changes, ReferenceEquals(strA, strB) = False
ReferenceEquals(stringC, strB) = False
stringC == strB = True
*/

```

The implementation of `Equals` in the [System.Object](#) universal base class also performs a reference equality

check, but it is best not to use this because, if a class happens to override the method, the results might not be what you expect. The same is true for the `==` and `!=` operators. When they are operating on reference types, the default behavior of `==` and `!=` is to perform a reference equality check. However, derived classes can overload the operator to perform a value equality check. To minimize the potential for error, it is best to always use [ReferenceEquals](#) when you have to determine whether two objects have reference equality.

Constant strings within the same assembly are always interned by the runtime. That is, only one instance of each unique literal string is maintained. However, the runtime does not guarantee that strings created at run time are interned, nor does it guarantee that two equal constant strings in different assemblies are interned.

See also

- [Equality Comparisons](#)

Casting and type conversions (C# Programming Guide)

12/28/2021 • 5 minutes to read • [Edit Online](#)

Because C# is statically-typed at compile time, after a variable is declared, it cannot be declared again or assigned a value of another type unless that type is implicitly convertible to the variable's type. For example, the `string` cannot be implicitly converted to `int`. Therefore, after you declare `i` as an `int`, you cannot assign the string "Hello" to it, as the following code shows:

```
int i;

// error CS0029: Cannot implicitly convert type 'string' to 'int'
i = "Hello";
```

However, you might sometimes need to copy a value into a variable or method parameter of another type. For example, you might have an integer variable that you need to pass to a method whose parameter is typed as `double`. Or you might need to assign a class variable to a variable of an interface type. These kinds of operations are called *type conversions*. In C#, you can perform the following kinds of conversions:

- **Implicit conversions:** No special syntax is required because the conversion always succeeds and no data will be lost. Examples include conversions from smaller to larger integral types, and conversions from derived classes to base classes.
- **Explicit conversions (casts):** Explicit conversions require a [cast expression](#). Casting is required when information might be lost in the conversion, or when the conversion might not succeed for other reasons. Typical examples include numeric conversion to a type that has less precision or a smaller range, and conversion of a base-class instance to a derived class.
- **User-defined conversions:** User-defined conversions are performed by special methods that you can define to enable explicit and implicit conversions between custom types that do not have a base class–derived class relationship. For more information, see [User-defined conversion operators](#).
- **Conversions with helper classes:** To convert between non-compatible types, such as integers and [System.DateTime](#) objects, or hexadecimal strings and byte arrays, you can use the [System.BitConverter](#) class, the [System.Convert](#) class, and the `Parse` methods of the built-in numeric types, such as [Int32.Parse](#). For more information, see [How to convert a byte array to an int](#), [How to convert a string to a number](#), and [How to convert between hexadecimal strings and numeric types](#).

Implicit conversions

For built-in numeric types, an implicit conversion can be made when the value to be stored can fit into the variable without being truncated or rounded off. For integral types, this means the range of the source type is a proper subset of the range for the target type. For example, a variable of type `long` (64-bit integer) can store any value that an `int` (32-bit integer) can store. In the following example, the compiler implicitly converts the value of `num` on the right to a type `long` before assigning it to `bigNum`.

```
// Implicit conversion. A long can
// hold any value an int can hold, and more!
int num = 2147483647;
long bigNum = num;
```

For a complete list of all implicit numeric conversions, see the [Implicit numeric conversions](#) section of the [Built-in numeric conversions](#) article.

For reference types, an implicit conversion always exists from a class to any one of its direct or indirect base classes or interfaces. No special syntax is necessary because a derived class always contains all the members of a base class.

```
Derived d = new Derived();

// Always OK.
Base b = d;
```

Explicit conversions

However, if a conversion cannot be made without a risk of losing information, the compiler requires that you perform an explicit conversion, which is called a *cast*. A cast is a way of explicitly informing the compiler that you intend to make the conversion and that you are aware that data loss might occur, or the cast may fail at run time. To perform a cast, specify the type that you are casting to in parentheses in front of the value or variable to be converted. The following program casts a [double](#) to an [int](#). The program will not compile without the cast.

```
class Test
{
    static void Main()
    {
        double x = 1234.7;
        int a;
        // Cast double to int.
        a = (int)x;
        System.Console.WriteLine(a);
    }
}
// Output: 1234
```

For a complete list of supported explicit numeric conversions, see the [Explicit numeric conversions](#) section of the [Built-in numeric conversions](#) article.

For reference types, an explicit cast is required if you need to convert from a base type to a derived type:

```
// Create a new derived type.
Giraffe g = new Giraffe();

// Implicit conversion to base type is safe.
Animal a = g;

// Explicit conversion is required to cast back
// to derived type. Note: This will compile but will
// throw an exception at run time if the right-side
// object is not in fact a Giraffe.
Giraffe g2 = (Giraffe)a;
```

A cast operation between reference types does not change the run-time type of the underlying object; it only changes the type of the value that is being used as a reference to that object. For more information, see

Type conversion exceptions at run time

In some reference type conversions, the compiler cannot determine whether a cast will be valid. It is possible for a cast operation that compiles correctly to fail at run time. As shown in the following example, a type cast that fails at run time will cause an [InvalidCastException](#) to be thrown.

```
class Animal
{
    public void Eat() => System.Console.WriteLine("Eating.");

    public override string ToString() => "I am an animal.";
}

class Reptile : Animal { }
class Mammal : Animal { }

class UnSafeCast
{
    static void Main()
    {
        Test(new Mammal());

        // Keep the console window open in debug mode.
        System.Console.WriteLine("Press any key to exit.");
        System.Console.ReadKey();
    }

    static void Test(Animal a)
    {
        // System.InvalidCastException at run time
        // Unable to cast object of type 'Mammal' to type 'Reptile'
        Reptile r = (Reptile)a;
    }
}
```

The `Test` method has an `Animal` parameter, thus explicitly casting the argument `a` to a `Reptile` makes a dangerous assumption. It is safer to not make assumptions, but rather check the type. C# provides the `is` operator to enable you to test for compatibility before actually performing a cast. For more information, see [How to safely cast using pattern matching and the `as` and `is` operators](#).

C# language specification

For more information, see the [Conversions](#) section of the [C# language specification](#).

See also

- [C# Programming Guide](#)
- [Types](#)
- [Cast expression](#)
- [User-defined conversion operators](#)
- [Generalized Type Conversion](#)
- [How to convert a string to a number](#)

Boxing and Unboxing (C# Programming Guide)

12/28/2021 • 5 minutes to read • [Edit Online](#)

Boxing is the process of converting a [value type](#) to the type `object` or to any interface type implemented by this value type. When the common language runtime (CLR) boxes a value type, it wraps the value inside a `System.Object` instance and stores it on the managed heap. Unboxing extracts the value type from the object. Boxing is implicit; unboxing is explicit. The concept of boxing and unboxing underlies the C# unified view of the type system in which a value of any type can be treated as an object.

In the following example, the integer variable `i` is *boxed* and assigned to object `o`.

```
int i = 123;
// The following line boxes i.
object o = i;
```

The object `o` can then be unboxed and assigned to integer variable `i`:

```
o = 123;
i = (int)o; // unboxing
```

The following examples illustrate how boxing is used in C#.

```
// String.Concat example.
// String.Concat has many versions. Rest the mouse pointer on
// Concat in the following statement to verify that the version
// that is used here takes three object arguments. Both 42 and
// true must be boxed.
Console.WriteLine(String.Concat("Answer", 42, true));

// List example.
// Create a list of objects to hold a heterogeneous collection
// of elements.
List<object> mixedList = new List<object>();

// Add a string element to the list.
mixedList.Add("First Group:");

// Add some integers to the list.
for (int j = 1; j < 5; j++)
{
    // Rest the mouse pointer over j to verify that you are adding
    // an int to a list of objects. Each element j is boxed when
    // you add j to mixedList.
    mixedList.Add(j);
}

// Add another string and more integers.
mixedList.Add("Second Group:");
for (int j = 5; j < 10; j++)
{
    mixedList.Add(j);
}

// Display the elements in the list. Declare the loop variable by
// using var, so that the compiler assigns its type.
foreach (var item in mixedList)
{
```

```

    // Rest the mouse pointer over item to verify that the elements
    // of mixedList are objects.
    Console.WriteLine(item);
}

// The following loop sums the squares of the first group of boxed
// integers in mixedList. The list elements are objects, and cannot
// be multiplied or added to the sum until they are unboxed. The
// unboxing must be done explicitly.
var sum = 0;
for (var j = 1; j < 5; j++)
{
    // The following statement causes a compiler error: Operator
    // '*' cannot be applied to operands of type 'object' and
    // 'object'.
    //sum += mixedList[j] * mixedList[j]);

    // After the list elements are unboxed, the computation does
    // not cause a compiler error.
    sum += (int)mixedList[j] * (int)mixedList[j];
}

// The sum displayed is 30, the sum of 1 + 4 + 9 + 16.
Console.WriteLine("Sum: " + sum);

// Output:
// Answer42True
// First Group:
// 1
// 2
// 3
// 4
// Second Group:
// 5
// 6
// 7
// 8
// 9
// Sum: 30

```

Performance

In relation to simple assignments, boxing and unboxing are computationally expensive processes. When a value type is boxed, a new object must be allocated and constructed. To a lesser degree, the cast required for unboxing is also expensive computationally. For more information, see [Performance](#).

Boxing

Boxing is used to store value types in the garbage-collected heap. Boxing is an implicit conversion of a [value type](#) to the type `object` or to any interface type implemented by this value type. Boxing a value type allocates an object instance on the heap and copies the value into the new object.

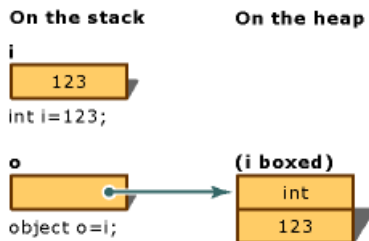
Consider the following declaration of a value-type variable:

```
int i = 123;
```

The following statement implicitly applies the boxing operation on the variable `i`:

```
// Boxing copies the value of i into object o.
object o = i;
```


The result of this statement is creating an object reference `o`, on the stack, that references a value of the type `int`, on the heap. This value is a copy of the value-type value assigned to the variable `i`. The difference between the two variables, `i` and `o`, is illustrated in the following image of boxing conversion:



It is also possible to perform the boxing explicitly as in the following example, but explicit boxing is never required:

```
int i = 123;
object o = (object)i; // explicit boxing
```

Example

This example converts an integer variable `i` to an object `o` by using boxing. Then, the value stored in the variable `i` is changed from 123 to 456. The example shows that the original value type and the boxed object use separate memory locations, and therefore can store different values.

```
class TestBoxing
{
    static void Main()
    {
        int i = 123;

        // Boxing copies the value of i into object o.
        object o = i;

        // Change the value of i.
        i = 456;

        // The change in i doesn't affect the value stored in o.
        System.Console.WriteLine("The value-type value = {0}", i);
        System.Console.WriteLine("The object-type value = {0}", o);
    }
}
/* Output:
    The value-type value = 456
    The object-type value = 123
*/
```

Unboxing

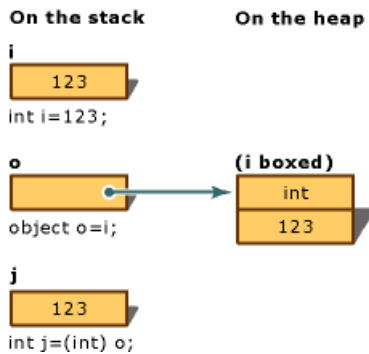
Unboxing is an explicit conversion from the type `object` to a [value type](#) or from an interface type to a value type that implements the interface. An unboxing operation consists of:

- Checking the object instance to make sure that it is a boxed value of the given value type.
- Copying the value from the instance into the value-type variable.

The following statements demonstrate both boxing and unboxing operations:

```
int i = 123;      // a value type
object o = i;    // boxing
int j = (int)o;  // unboxing
```

The following figure demonstrates the result of the previous statements:



For the unboxing of value types to succeed at run time, the item being unboxed must be a reference to an object that was previously created by boxing an instance of that value type. Attempting to unbox `null` causes a [NullReferenceException](#). Attempting to unbox a reference to an incompatible value type causes an [InvalidCastException](#).

Example

The following example demonstrates a case of invalid unboxing and the resulting `InvalidCastException`. Using `try` and `catch`, an error message is displayed when the error occurs.

```
class TestUnboxing
{
    static void Main()
    {
        int i = 123;
        object o = i; // implicit boxing

        try
        {
            int j = (short)o; // attempt to unbox

            System.Console.WriteLine("Unboxing OK.");
        }
        catch (System.InvalidCastException e)
        {
            System.Console.WriteLine("{0} Error: Incorrect unboxing.", e.Message);
        }
    }
}
```

This program outputs:

```
Specified cast is not valid. Error: Incorrect unboxing.
```

If you change the statement:

```
int j = (short)o;
```

to:

```
int j = (int)o;
```

the conversion will be performed, and you will get the output:

```
Unboxing OK.
```

C# language specification

For more information, see the [C# Language Specification](#). The language specification is the definitive source for C# syntax and usage.

See also

- [C# programming guide](#)
- [Reference types](#)
- [Value types](#)

How to convert a byte array to an int (C# Programming Guide)

12/28/2021 • 2 minutes to read • [Edit Online](#)

This example shows you how to use the [BitConverter](#) class to convert an array of bytes to an [int](#) and back to an array of bytes. You may have to convert from bytes to a built-in data type after you read bytes off the network, for example. In addition to the [ToInt32\(Byte\[\], Int32\)](#) method in the example, the following table lists methods in the [BitConverter](#) class that convert bytes (from an array of bytes) to other built-in types.

TYPE RETURNED	METHOD
<code>bool</code>	ToBoolean(Byte[], Int32)
<code>char</code>	ToChar(Byte[], Int32)
<code>double</code>	ToDouble(Byte[], Int32)
<code>short</code>	ToInt16(Byte[], Int32)
<code>int</code>	ToInt32(Byte[], Int32)
<code>long</code>	ToInt64(Byte[], Int32)
<code>float</code>	ToSingle(Byte[], Int32)
<code>ushort</code>	ToUInt16(Byte[], Int32)
<code>uint</code>	ToUInt32(Byte[], Int32)
<code>ulong</code>	ToUInt64(Byte[], Int32)

Examples

This example initializes an array of bytes, reverses the array if the computer architecture is little-endian (that is, the least significant byte is stored first), and then calls the [ToInt32\(Byte\[\], Int32\)](#) method to convert four bytes in the array to an `int`. The second argument to [ToInt32\(Byte\[\], Int32\)](#) specifies the start index of the array of bytes.

NOTE

The output may differ depending on the endianness of your computer's architecture.

```
byte[] bytes = { 0, 0, 0, 25 };

// If the system architecture is little-endian (that is, little end first),
// reverse the byte array.
if (BitConverter.IsLittleEndian)
    Array.Reverse(bytes);

int i = BitConverter.ToInt32(bytes, 0);
Console.WriteLine("int: {0}", i);
// Output: int: 25
```

In this example, the [GetBytes\(Int32\)](#) method of the [BitConverter](#) class is called to convert an `int` to an array of bytes.

NOTE

The output may differ depending on the endianness of your computer's architecture.

```
byte[] bytes = BitConverter.GetBytes(201805978);
Console.WriteLine("byte array: " + BitConverter.ToString(bytes));
// Output: byte array: 9A-50-07-0C
```

See also

- [BitConverter](#)
- [IsLittleEndian](#)
- [Types](#)

How to convert a string to a number (C# Programming Guide)

12/28/2021 • 4 minutes to read • [Edit Online](#)

You convert a `string` to a number by calling the `Parse` or `TryParse` method found on numeric types (`int`, `long`, `double`, and so on), or by using methods in the [System.Convert](#) class.

It's slightly more efficient and straightforward to call a `TryParse` method (for example, `int.TryParse("11", out number)`) or `Parse` method (for example, `var number = int.Parse("11")`). Using a [Convert](#) method is more useful for general objects that implement [IConvertible](#).

You use `Parse` or `TryParse` methods on the numeric type you expect the string contains, such as the [System.Int32](#) type. The [Convert.ToInt32](#) method uses [Parse](#) internally. The `Parse` method returns the converted number; the `TryParse` method returns a boolean value that indicates whether the conversion succeeded, and returns the converted number in an `out` parameter. If the string isn't in a valid format, `Parse` throws an exception, but `TryParse` returns `false`. When calling a `Parse` method, you should always use exception handling to catch a [FormatException](#) when the parse operation fails.

Call Parse or TryParse methods

The `Parse` and `TryParse` methods ignore white space at the beginning and at the end of the string, but all other characters must be characters that form the appropriate numeric type (`int`, `long`, `ulong`, `float`, `decimal`, and so on). Any white space within the string that forms the number causes an error. For example, you can use `decimal.TryParse` to parse "10", "10.3", or " 10 ", but you can't use this method to parse 10 from "10X", "1 0" (note the embedded space), "10 .3" (note the embedded space), "10e1" (`float.TryParse` works here), and so on. A string whose value is `null` or [String.Empty](#) fails to parse successfully. You can check for a null or empty string before attempting to parse it by calling the [String.IsNullOrEmpty](#) method.

The following example demonstrates both successful and unsuccessful calls to `Parse` and `TryParse`.

```

using System;

public static class StringConversion
{
    public static void Main()
    {
        string input = String.Empty;
        try
        {
            int result = Int32.Parse(input);
            Console.WriteLine(result);
        }
        catch (FormatException)
        {
            Console.WriteLine($"Unable to parse '{input}'");
        }
        // Output: Unable to parse ''

        try
        {
            int numVal = Int32.Parse("-105");
            Console.WriteLine(numVal);
        }
        catch (FormatException e)
        {
            Console.WriteLine(e.Message);
        }
        // Output: -105

        if (Int32.TryParse("-105", out int j))
        {
            Console.WriteLine(j);
        }
        else
        {
            Console.WriteLine("String could not be parsed.");
        }
        // Output: -105

        try
        {
            int m = Int32.Parse("abc");
        }
        catch (FormatException e)
        {
            Console.WriteLine(e.Message);
        }
        // Output: Input string was not in a correct format.

        const string inputString = "abc";
        if (Int32.TryParse(inputString, out int numValue))
        {
            Console.WriteLine(numValue);
        }
        else
        {
            Console.WriteLine($"Int32.TryParse could not parse '{inputString}' to an int.");
        }
        // Output: Int32.TryParse could not parse 'abc' to an int.
    }
}

```

The following example illustrates one approach to parsing a string expected to include leading numeric characters (including hexadecimal characters) and trailing non-numeric characters. It assigns valid characters from the beginning of a string to a new string before calling the [TryParse](#) method. Because the strings to be

parsed contain a few characters, the example calls the [String.Concat](#) method to assign valid characters to a new string. For a larger string, the [StringBuilder](#) class can be used instead.

```
using System;

public static class StringConversion
{
    public static void Main()
    {
        var str = " 10FFxxx";
        string numericString = string.Empty;
        foreach (var c in str)
        {
            // Check for numeric characters (hex in this case) or leading or trailing spaces.
            if ((c >= '0' && c <= '9') || (char.ToUpperInvariant(c) >= 'A' && char.ToUpperInvariant(c) <=
            'F') || c == ' ')
            {
                numericString = string.Concat(numericString, c.ToString());
            }
            else
            {
                break;
            }
        }

        if (int.TryParse(numericString, System.Globalization.NumberStyles.HexNumber, null, out int i))
        {
            Console.WriteLine($"'{str}' --> '{numericString}' --> {i}");
        }
        // Output: ' 10FFxxx' --> ' 10FF' --> 4351

        str = " -10FFXXX";
        numericString = "";
        foreach (char c in str)
        {
            // Check for numeric characters (0-9), a negative sign, or leading or trailing spaces.
            if ((c >= '0' && c <= '9') || c == ' ' || c == '-')
            {
                numericString = string.Concat(numericString, c);
            }
            else
            {
                break;
            }
        }

        if (int.TryParse(numericString, out int j))
        {
            Console.WriteLine($"'{str}' --> '{numericString}' --> {j}");
        }
        // Output: ' -10FFXXX' --> ' -10' --> -10
    }
}
```

Call Convert methods

The following table lists some of the methods from the [Convert](#) class that you can use to convert a string to a number.

NUMERIC TYPE	METHOD
<code>decimal</code>	ToDecimal(String)

NUMERIC TYPE	METHOD
<code>float</code>	ToSingle(String)
<code>double</code>	ToDouble(String)
<code>short</code>	ToInt16(String)
<code>int</code>	ToInt32(String)
<code>long</code>	ToInt64(String)
<code>ushort</code>	ToUInt16(String)
<code>uint</code>	ToUInt32(String)
<code>ulong</code>	ToUInt64(String)

The following example calls the [Convert.ToInt32\(String\)](#) method to convert an input string to an `int`. The example catches the two most common exceptions that can be thrown by this method, [FormatException](#) and [OverflowException](#). If the resulting number can be incremented without exceeding [Int32.MaxValue](#), the example adds 1 to the result and displays the output.

```

using System;

public class ConvertStringExample1
{
    static void Main(string[] args)
    {
        int numVal = -1;
        bool repeat = true;

        while (repeat)
        {
            Console.Write("Enter a number between -2,147,483,648 and +2,147,483,647 (inclusive): ");

            string input = Console.ReadLine();

            // ToInt32 can throw FormatException or OverflowException.
            try
            {
                numVal = Convert.ToInt32(input);
                if (numVal < Int32.MaxValue)
                {
                    Console.WriteLine("The new value is {0}", ++numVal);
                }
                else
                {
                    Console.WriteLine("numVal cannot be incremented beyond its current value");
                }
            }
            catch (FormatException)
            {
                Console.WriteLine("Input string is not a sequence of digits.");
            }
            catch (OverflowException)
            {
                Console.WriteLine("The number cannot fit in an Int32.");
            }

            Console.Write("Go again? Y/N: ");
            string go = Console.ReadLine();
            if (go.ToUpper() != "Y")
            {
                repeat = false;
            }
        }
    }
}

// Sample Output:
// Enter a number between -2,147,483,648 and +2,147,483,647 (inclusive): 473
// The new value is 474
// Go again? Y/N: y
// Enter a number between -2,147,483,648 and +2,147,483,647 (inclusive): 2147483647
// numVal cannot be incremented beyond its current value
// Go again? Y/N: y
// Enter a number between -2,147,483,648 and +2,147,483,647 (inclusive): -1000
// The new value is -999
// Go again? Y/N: n

```

How to convert between hexadecimal strings and numeric types (C# Programming Guide)

12/28/2021 • 3 minutes to read • [Edit Online](#)

These examples show you how to perform the following tasks:

- Obtain the hexadecimal value of each character in a `string`.
- Obtain the `char` that corresponds to each value in a hexadecimal string.
- Convert a hexadecimal `string` to an `int`.
- Convert a hexadecimal `string` to a `float`.
- Convert a `byte` array to a hexadecimal `string`.

Examples

This example outputs the hexadecimal value of each character in a `string`. First it parses the `string` to an array of characters. Then it calls `ToInt32(Char)` on each character to obtain its numeric value. Finally, it formats the number as its hexadecimal representation in a `string`.

```
string input = "Hello World!";
char[] values = input.ToCharArray();
foreach (char letter in values)
{
    // Get the integral value of the character.
    int value = Convert.ToInt32(letter);
    // Convert the integer value to a hexadecimal value in string form.
    Console.WriteLine($"Hexadecimal value of {letter} is {value:X}");
}
/* Output:
Hexadecimal value of H is 48
Hexadecimal value of e is 65
Hexadecimal value of l is 6C
Hexadecimal value of l is 6C
Hexadecimal value of o is 6F
Hexadecimal value of   is 20
Hexadecimal value of W is 57
Hexadecimal value of o is 6F
Hexadecimal value of r is 72
Hexadecimal value of l is 6C
Hexadecimal value of d is 64
Hexadecimal value of ! is 21
*/
```

This example parses a `string` of hexadecimal values and outputs the character corresponding to each hexadecimal value. First it calls the `Split(Char[])` method to obtain each hexadecimal value as an individual `string` in an array. Then it calls `ToInt32(String, Int32)` to convert the hexadecimal value to a decimal value represented as an `int`. It shows two different ways to obtain the character corresponding to that character code. The first technique uses `ConvertFromUtf32(Int32)`, which returns the character corresponding to the integer argument as a `string`. The second technique explicitly casts the `int` to a `char`.

```

string hexValues = "48 65 6C 6C 6F 20 57 6F 72 6C 64 21";
string[] hexValuesSplit = hexValues.Split(' ');
foreach (string hex in hexValuesSplit)
{
    // Convert the number expressed in base-16 to an integer.
    int value = Convert.ToInt32(hex, 16);
    // Get the character corresponding to the integral value.
    string stringValue = Char.ConvertFromUtf32(value);
    char charValue = (char)value;
    Console.WriteLine("hexadecimal value = {0}, int value = {1}, char value = {2} or {3}",
        hex, value, stringValue, charValue);
}
/* Output:
    hexadecimal value = 48, int value = 72, char value = H or H
    hexadecimal value = 65, int value = 101, char value = e or e
    hexadecimal value = 6C, int value = 108, char value = l or l
    hexadecimal value = 6C, int value = 108, char value = l or l
    hexadecimal value = 6F, int value = 111, char value = o or o
    hexadecimal value = 20, int value = 32, char value =   or
    hexadecimal value = 57, int value = 87, char value = W or W
    hexadecimal value = 6F, int value = 111, char value = o or o
    hexadecimal value = 72, int value = 114, char value = r or r
    hexadecimal value = 6C, int value = 108, char value = l or l
    hexadecimal value = 64, int value = 100, char value = d or d
    hexadecimal value = 21, int value = 33, char value = ! or !
*/

```

This example shows another way to convert a hexadecimal `string` to an integer, by calling the [Parse\(String, NumberStyles\)](#) method.

```

string hexString = "8E2";
int num = Int32.Parse(hexString, System.Globalization.NumberStyles.HexNumber);
Console.WriteLine(num);
//Output: 2274

```

The following example shows how to convert a hexadecimal `string` to a `float` by using the [System.BitConverter](#) class and the [UInt32.Parse](#) method.

```

string hexString = "43480170";
uint num = uint.Parse(hexString, System.Globalization.NumberStyles.AllowHexSpecifier);

byte[] floatVals = BitConverter.GetBytes(num);
float f = BitConverter.ToSingle(floatVals, 0);
Console.WriteLine("float convert = {0}", f);

// Output: 200.0056

```

The following example shows how to convert a `byte` array to a hexadecimal string by using the [System.BitConverter](#) class.

```

byte[] vals = { 0x01, 0xAA, 0xB1, 0xDC, 0x10, 0xDD };

string str = BitConverter.ToString(vals);
Console.WriteLine(str);

str = BitConverter.ToString(vals).Replace("-", "");
Console.WriteLine(str);

/*Output:
    01-AA-B1-DC-10-DD
    01AAB1DC10DD
*/

```

The following example shows how to convert a [byte](#) array to a hexadecimal string by calling the [Convert.ToHexString](#) method introduced in .NET 5.0.

```

byte[] array = { 0x64, 0x6f, 0x74, 0x63, 0x65, 0x74 };

string hexValue = Convert.ToHexString(array);
Console.WriteLine(hexValue);

/*Output:
    646F74636574
*/

```

See also

- [Standard Numeric Format Strings](#)
- [Types](#)
- [How to determine whether a string represents a numeric value](#)

Using type dynamic (C# Programming Guide)

12/28/2021 • 5 minutes to read • [Edit Online](#)

C# 4 introduces a new type, `dynamic`. The type is a static type, but an object of type `dynamic` bypasses static type checking. In most cases, it functions like it has type `object`. At compile time, an element that is typed as `dynamic` is assumed to support any operation. Therefore, you do not have to be concerned about whether the object gets its value from a COM API, from a dynamic language such as IronPython, from the HTML Document Object Model (DOM), from reflection, or from somewhere else in the program. However, if the code is not valid, errors are caught at run time.

For example, if instance method `exampleMethod1` in the following code has only one parameter, the compiler recognizes that the first call to the method, `ec.exampleMethod1(10, 4)`, is not valid because it contains two arguments. The call causes a compiler error. The second call to the method, `dynamic_ec.exampleMethod1(10, 4)`, is not checked by the compiler because the type of `dynamic_ec` is `dynamic`. Therefore, no compiler error is reported. However, the error does not escape notice indefinitely. It is caught at run time and causes a run-time exception.

```
static void Main(string[] args)
{
    ExampleClass ec = new ExampleClass();
    // The following call to exampleMethod1 causes a compiler error
    // if exampleMethod1 has only one parameter. Uncomment the line
    // to see the error.
    //ec.exampleMethod1(10, 4);

    dynamic dynamic_ec = new ExampleClass();
    // The following line is not identified as an error by the
    // compiler, but it causes a run-time exception.
    dynamic_ec.exampleMethod1(10, 4);

    // The following calls also do not cause compiler errors, whether
    // appropriate methods exist or not.
    dynamic_ec.someMethod("some argument", 7, null);
    dynamic_ec.nonexistentMethod();
}
```

```
class ExampleClass
{
    public ExampleClass() { }
    public ExampleClass(int v) { }

    public void exampleMethod1(int i) { }

    public void exampleMethod2(string str) { }
}
```

The role of the compiler in these examples is to package together information about what each statement is proposing to do to the object or expression that is typed as `dynamic`. At run time, the stored information is examined, and any statement that is not valid causes a run-time exception.

The result of most dynamic operations is itself `dynamic`. For example, if you rest the mouse pointer over the use of `testSum` in the following example, IntelliSense displays the type (local variable) `dynamic testSum`.

```
dynamic d = 1;
var testSum = d + 3;
// Rest the mouse pointer over testSum in the following statement.
System.Console.WriteLine(testSum);
```

Operations in which the result is not `dynamic` include:

- Conversions from `dynamic` to another type.
- Constructor calls that include arguments of type `dynamic`.

For example, the type of `testInstance` in the following declaration is `ExampleClass`, not `dynamic`:

```
var testInstance = new ExampleClass(d);
```

Conversion examples are shown in the following section, "Conversions."

Conversions

Conversions between dynamic objects and other types are easy. This enables the developer to switch between dynamic and non-dynamic behavior.

Any object can be converted to dynamic type implicitly, as shown in the following examples.

```
dynamic d1 = 7;
dynamic d2 = "a string";
dynamic d3 = System.DateTime.Today;
dynamic d4 = System.Diagnostics.Process.GetProcesses();
```

Conversely, an implicit conversion can be dynamically applied to any expression of type `dynamic`.

```
int i = d1;
string str = d2;
DateTime dt = d3;
System.Diagnostics.Process[] procs = d4;
```

Overload resolution with arguments of type dynamic

Overload resolution occurs at run time instead of at compile time if one or more of the arguments in a method call have the type `dynamic`, or if the receiver of the method call is of type `dynamic`. In the following example, if the only accessible `exampleMethod2` method is defined to take a string argument, sending `d1` as the argument does not cause a compiler error, but it does cause a run-time exception. Overload resolution fails at run time because the run-time type of `d1` is `int`, and `exampleMethod2` requires a string.

```
// Valid.
ec.exampleMethod2("a string");

// The following statement does not cause a compiler error, even though ec is not
// dynamic. A run-time exception is raised because the run-time type of d1 is int.
ec.exampleMethod2(d1);
// The following statement does cause a compiler error.
//ec.exampleMethod2(7);
```

Dynamic language runtime

The dynamic language runtime (DLR) is an API that was introduced in .NET Framework 4. It provides the infrastructure that supports the `dynamic` type in C#, and also the implementation of dynamic programming languages such as IronPython and IronRuby. For more information about the DLR, see [Dynamic Language Runtime Overview](#).

COM interop

C# 4 includes several features that improve the experience of interoperating with COM APIs such as the Office Automation APIs. Among the improvements are the use of the `dynamic` type, and of [named and optional arguments](#).

Many COM methods allow for variation in argument types and return type by designating the types as `object`. This has necessitated explicit casting of the values to coordinate with strongly typed variables in C#. If you compile by using the [EmbedInteropTypes \(C# Compiler Options\)](#) option, the introduction of the `dynamic` type enables you to treat the occurrences of `object` in COM signatures as if they were of type `dynamic`, and thereby to avoid much of the casting. For example, the following statements contrast how you access a cell in a Microsoft Office Excel spreadsheet with the `dynamic` type and without the `dynamic` type.

```
// Before the introduction of dynamic.
((Excel.Range)excelApp.Cells[1, 1]).Value2 = "Name";
Excel.Range range2008 = (Excel.Range)excelApp.Cells[1, 1];
```

```
// After the introduction of dynamic, the access to the Value property and
// the conversion to Excel.Range are handled by the run-time COM binder.
excelApp.Cells[1, 1].Value = "Name";
Excel.Range range2010 = excelApp.Cells[1, 1];
```

Related topics

TITLE	DESCRIPTION
dynamic	Describes the usage of the <code>dynamic</code> keyword.
Dynamic Language Runtime Overview	Provides an overview of the DLR, which is a runtime environment that adds a set of services for dynamic languages to the common language runtime (CLR).
Walkthrough: Creating and Using Dynamic Objects	Provides step-by-step instructions for creating a custom dynamic object and for creating a project that accesses an <code>IronPython</code> library.
How to access Office interop objects by using C# features	Demonstrates how to create a project that uses named and optional arguments, the <code>dynamic</code> type, and other enhancements that simplify access to Office API objects.

Walkthrough: Creating and Using Dynamic Objects (C# and Visual Basic)

12/28/2021 • 11 minutes to read • [Edit Online](#)

Dynamic objects expose members such as properties and methods at run time, instead of at compile time. This enables you to create objects to work with structures that do not match a static type or format. For example, you can use a dynamic object to reference the HTML Document Object Model (DOM), which can contain any combination of valid HTML markup elements and attributes. Because each HTML document is unique, the members for a particular HTML document are determined at run time. A common method to reference an attribute of an HTML element is to pass the name of the attribute to the `GetProperty` method of the element. To reference the `id` attribute of the HTML element `<div id="Div1">`, you first obtain a reference to the `<div>` element, and then use `divElement.GetProperty("id")`. If you use a dynamic object, you can reference the `id` attribute as `divElement.id`.

Dynamic objects also provide convenient access to dynamic languages such as IronPython and IronRuby. You can use a dynamic object to refer to a dynamic script that is interpreted at run time.

You reference a dynamic object by using late binding. In C#, you specify the type of a late-bound object as `dynamic`. In Visual Basic, you specify the type of a late-bound object as `Object`. For more information, see [dynamic](#) and [Early and Late Binding](#).

You can create custom dynamic objects by using the classes in the `System.Dynamic` namespace. For example, you can create an `ExpandoObject` and specify the members of that object at run time. You can also create your own type that inherits the `DynamicObject` class. You can then override the members of the `DynamicObject` class to provide run-time dynamic functionality.

This article contains two independent walkthroughs:

- Create a custom object that dynamically exposes the contents of a text file as properties of an object.
- Create a project that uses an `IronPython` library.

You can do either one of these or both of them, and if you do both, the order doesn't matter.

Prerequisites

- [Visual Studio 2019 version 16.9 or a later version](#) with the **.NET desktop development** workload installed. The .NET 5 SDK is automatically installed when you select this workload.

NOTE

Your computer might show different names or locations for some of the Visual Studio user interface elements in the following instructions. The Visual Studio edition that you have and the settings that you use determine these elements. For more information, see [Personalizing the IDE](#).

- For the second walkthrough, install [IronPython](#) for .NET. Go to their [Download page](#) to obtain the latest version.

Create a Custom Dynamic Object

The first walkthrough defines a custom dynamic object that searches the contents of a text file. A dynamic

property specifies the text to search for. For example, if calling code specifies `dynamicFile.Sample`, the dynamic class returns a generic list of strings that contains all of the lines from the file that begin with "Sample". The search is case-insensitive. The dynamic class also supports two optional arguments. The first argument is a search option enum value that specifies that the dynamic class should search for matches at the start of the line, the end of the line, or anywhere in the line. The second argument specifies that the dynamic class should trim leading and trailing spaces from each line before searching. For example, if calling code specifies `dynamicFile.Sample(StringSearchOption.Contains)`, the dynamic class searches for "Sample" anywhere in a line. If calling code specifies `dynamicFile.Sample(StringSearchOption.StartsWith, false)`, the dynamic class searches for "Sample" at the start of each line, and does not remove leading and trailing spaces. The default behavior of the dynamic class is to search for a match at the start of each line and to remove leading and trailing spaces.

To create a custom dynamic class

1. Start Visual Studio.
2. Select **Create a new project**.
3. In the **Create a new project** dialog, select C# or Visual Basic, select **Console Application**, and then select **Next**.
4. In the **Configure your new project** dialog, enter `DynamicSample` for the **Project name**, and then select **Next**.
5. In the **Additional information** dialog, select **.NET 5.0 (Current)** for the **Target Framework**, and then select **Create**.

The new project is created.

6. In **Solution Explorer**, right-click the `DynamicSample` project and select **Add > Class**. In the **Name** box, type `ReadOnlyFile`, and then select **Add**.

A new file is added that contains the `ReadOnlyFile` class.

7. At the top of the `ReadOnlyFile.cs` or `ReadOnlyFile.vb` file, add the following code to import the [System.IO](#) and [System.Dynamic](#) namespaces.

```
using System.IO;
using System.Dynamic;
```

```
Imports System.IO
Imports System.Dynamic
```

8. The custom dynamic object uses an enum to determine the search criteria. Before the class statement, add the following enum definition.

```
public enum StringSearchOption
{
    StartsWith,
    Contains,
    EndsWith
}
```

```
Public Enum StringSearchOption
    StartsWith
    Contains
    EndsWith
End Enum
```

9. Update the class statement to inherit the `DynamicObject` class, as shown in the following code example.

```
class ReadOnlyFile : DynamicObject
```

```
Public Class ReadOnlyFile
    Inherits DynamicObject
```

10. Add the following code to the `ReadOnlyFile` class to define a private field for the file path and a constructor for the `ReadOnlyFile` class.

```
// Store the path to the file and the initial line count value.
private string p_filePath;

// Public constructor. Verify that file exists and store the path in
// the private variable.
public ReadOnlyFile(string filePath)
{
    if (!File.Exists(filePath))
    {
        throw new Exception("File path does not exist.");
    }

    p_filePath = filePath;
}
```

```
' Store the path to the file and the initial line count value.
Private p_filePath As String

' Public constructor. Verify that file exists and store the path in
' the private variable.
Public Sub New(ByVal filePath As String)
    If Not File.Exists(filePath) Then
        Throw New Exception("File path does not exist.")
    End If

    p_filePath = filePath
End Sub
```

11. Add the following `GetPropertyValue` method to the `ReadOnlyFile` class. The `GetPropertyValue` method takes, as input, search criteria and returns the lines from a text file that match that search criteria. The dynamic methods provided by the `ReadOnlyFile` class call the `GetPropertyValue` method to retrieve their respective results.

```

public List<string> GetPropertyValue(string propertyName,
                                   StringSearchOption StringSearchOption =
StringSearchOption.StartsWith,
                                   bool trimSpaces = true)
{
    StreamReader sr = null;
    List<string> results = new List<string>();
    string line = "";
    string testLine = "";

    try
    {
        sr = new StreamReader(p_filePath);

        while (!sr.EndOfStream)
        {
            line = sr.ReadLine();

            // Perform a case-insensitive search by using the specified search options.
            testLine = line.ToUpper();
            if (trimSpaces) { testLine = testLine.Trim(); }

            switch (StringSearchOption)
            {
                case StringSearchOption.StartsWith:
                    if (testLine.StartsWith(propertyName.ToUpper())) { results.Add(line); }
                    break;
                case StringSearchOption.Contains:
                    if (testLine.Contains(propertyName.ToUpper())) { results.Add(line); }
                    break;
                case StringSearchOption.EndsWith:
                    if (testLine.EndsWith(propertyName.ToUpper())) { results.Add(line); }
                    break;
            }
        }
    }
    catch
    {
        // Trap any exception that occurs in reading the file and return null.
        results = null;
    }
    finally
    {
        if (sr != null) {sr.Close();}
    }

    return results;
}

```

```

Public Function GetPropertyValue(ByVal propertyName As String,
                                Optional ByVal searchStringOption As StringSearchOption =
StringSearchOption.StartsWith,
                                Optional ByVal trimSpaces As Boolean = True) As List(Of String)

    Dim sr As StreamReader = Nothing
    Dim results As New List(Of String)
    Dim line = ""
    Dim testLine = ""

    Try
        sr = New StreamReader(p_filePath)

        While Not sr.EndOfStream
            line = sr.ReadLine()

            ' Perform a case-insensitive search by using the specified search options.
            testLine = UCase(line)
            If trimSpaces Then testLine = Trim(testLine)

            Select Case searchStringOption
                Case StringSearchOption.StartsWith
                    If testLine.StartsWith(UCase(propertyName)) Then results.Add(line)
                Case StringSearchOption.Contains
                    If testLine.Contains(UCase(propertyName)) Then results.Add(line)
                Case StringSearchOption.EndsWith
                    If testLine.EndsWith(UCase(propertyName)) Then results.Add(line)
            End Select
        End While
    Catch
        ' Trap any exception that occurs in reading the file and return Nothing.
        results = Nothing
    Finally
        If sr IsNot Nothing Then sr.Close()
    End Try

    Return results
End Function

```

12. After the `GetPropertyValue` method, add the following code to override the `TryGetMember` method of the `DynamicObject` class. The `TryGetMember` method is called when a member of a dynamic class is requested and no arguments are specified. The `binder` argument contains information about the referenced member, and the `result` argument references the result returned for the specified member. The `TryGetMember` method returns a Boolean value that returns `true` if the requested member exists; otherwise it returns `false`.

```

// Implement the TryGetMember method of the DynamicObject class for dynamic member calls.
public override bool TryGetMember(GetMemberBinder binder,
                                out object result)

{
    result = GetPropertyValue(binder.Name);
    return result == null ? false : true;
}

```

```

' Implement the TryGetMember method of the DynamicObject class for dynamic member calls.
Public Overrides Function TryGetMember(ByVal binder As GetMemberBinder,
                                       ByRef result As Object) As Boolean

    result = GetPropertyValue(binder.Name)
    Return If(result Is Nothing, False, True)
End Function

```

13. After the `TryGetMember` method, add the following code to override the `TryInvokeMember` method of the `DynamicObject` class. The `TryInvokeMember` method is called when a member of a dynamic class is requested with arguments. The `binder` argument contains information about the referenced member, and the `result` argument references the result returned for the specified member. The `args` argument contains an array of the arguments that are passed to the member. The `TryInvokeMember` method returns a Boolean value that returns `true` if the requested member exists; otherwise it returns `false`.

The custom version of the `TryInvokeMember` method expects the first argument to be a value from the `StringSearchOption` enum that you defined in a previous step. The `TryInvokeMember` method expects the second argument to be a Boolean value. If one or both arguments are valid values, they are passed to the `GetProperty` method to retrieve the results.

```
// Implement the TryInvokeMember method of the DynamicObject class for
// dynamic member calls that have arguments.
public override bool TryInvokeMember(InvokeMemberBinder binder,
                                     object[] args,
                                     out object result)
{
    StringSearchOption StringSearchOption = StringSearchOption.StartsWith;
    bool trimSpaces = true;

    try
    {
        if (args.Length > 0) { StringSearchOption = (StringSearchOption)args[0]; }
    }
    catch
    {
        throw new ArgumentException("StringSearchOption argument must be a StringSearchOption enum
value.");
    }

    try
    {
        if (args.Length > 1) { trimSpaces = (bool)args[1]; }
    }
    catch
    {
        throw new ArgumentException("trimSpaces argument must be a Boolean value.");
    }

    result = GetPropertyValue(binder.Name, StringSearchOption, trimSpaces);

    return result == null ? false : true;
}
```

```

' Implement the TryInvokeMember method of the DynamicObject class for
' dynamic member calls that have arguments.
Public Overrides Function TryInvokeMember(ByVal binder As InvokeMemberBinder,
                                         ByVal args() As Object,
                                         ByRef result As Object) As Boolean

    Dim StringSearchOption As StringSearchOption = StringSearchOption.StartsWith
    Dim trimSpaces = True

    Try
        If args.Length > 0 Then StringSearchOption = CType(args(0), StringSearchOption)
    Catch
        Throw New ArgumentException("StringSearchOption argument must be a StringSearchOption enum
value.")
    End Try

    Try
        If args.Length > 1 Then trimSpaces = CType(args(1), Boolean)
    Catch
        Throw New ArgumentException("trimSpaces argument must be a Boolean value.")
    End Try

    result = GetPropertyValue(binder.Name, StringSearchOption, trimSpaces)

    Return If(result Is Nothing, False, True)
End Function

```

14. Save and close the file.

To create a sample text file

1. In **Solution Explorer**, right-click the **DynamicSample** project and select **Add > New Item**. In the **Installed Templates** pane, select **General**, and then select the **Text File** template. Leave the default name of *TextFile1.txt* in the **Name** box, and then click **Add**. A new text file is added to the project.
2. Copy the following text to the *TextFile1.txt* file.

```

List of customers and suppliers

Supplier: Lucerne Publishing (https://www.lucernepublishing.com/)
Customer: Preston, Chris
Customer: Hines, Patrick
Customer: Cameron, Maria
Supplier: Graphic Design Institute (https://www.graphicdesigninstitute.com/)
Supplier: Fabrikam, Inc. (https://www.fabrikam.com/)
Customer: Seubert, Roxanne
Supplier: Proseware, Inc. (http://www.proseware.com/)
Customer: Adolphi, Stephan
Customer: Koch, Paul

```

3. Save and close the file.

To create a sample application that uses the custom dynamic object

1. In **Solution Explorer**, double-click the *Program.vb* file if you're using Visual Basic or the *Program.cs* file if you're using Visual C#.
2. Add the following code to the **Main** procedure to create an instance of the **ReadOnlyFile** class for the *TextFile1.txt* file. The code uses late binding to call dynamic members and retrieve lines of text that contain the string "Customer".

```
dynamic rFile = new ReadOnlyFile(@"..\..\..\TextFile1.txt");
foreach (string line in rFile.Customer)
{
    Console.WriteLine(line);
}
Console.WriteLine("-----");
foreach (string line in rFile.Customer(StringSearchOption.Contains, true))
{
    Console.WriteLine(line);
}
```

```
Dim rFile As Object = New ReadOnlyFile("../..\..\TextFile1.txt")
For Each line In rFile.Customer
    Console.WriteLine(line)
Next
Console.WriteLine("-----")
For Each line In rFile.Customer(StringSearchOption.Contains, True)
    Console.WriteLine(line)
Next
```

3. Save the file and press **Ctrl+F5** to build and run the application.

Call a dynamic language library

The following walkthrough creates a project that accesses a library that is written in the dynamic language IronPython.

To create a custom dynamic class

1. In Visual Studio, select **File > New > Project**.
2. In the **Create a new project** dialog, select **C#** or **Visual Basic**, select **Console Application**, and then select **Next**.
3. In the **Configure your new project** dialog, enter `DynamicIronPythonSample` for the **Project name**, and then select **Next**.
4. In the **Additional information** dialog, select **.NET 5.0 (Current)** for the **Target Framework**, and then select **Create**.

The new project is created.

5. Install the [IronPython](#) NuGet package.
6. If you're using Visual Basic, edit the *Program.vb* file. If you're using Visual C#, edit the *Program.cs* file.
7. At the top of the file, add the following code to import the `Microsoft.Scripting.Hosting` and `IronPython.Hosting` namespaces from the IronPython libraries and the `System.Linq` namespace.

```
using System.Linq;
using Microsoft.Scripting.Hosting;
using IronPython.Hosting;
```

```
Imports Microsoft.Scripting.Hosting
Imports IronPython.Hosting
Imports System.Linq
```

8. In the **Main** method, add the following code to create a new `Microsoft.Scripting.Hosting.ScriptRuntime`

object to host the IronPython libraries. The `ScriptRuntime` object loads the IronPython library module `random.py`.

```
// Set the current directory to the IronPython libraries.
System.IO.Directory.SetCurrentDirectory(
    Environment.GetFolderPath(Environment.SpecialFolder.ProgramFiles) +
    @"IronPython 2.7\Lib");

// Create an instance of the random.py IronPython library.
Console.WriteLine("Loading random.py");
ScriptRuntime py = Python.CreateRuntime();
dynamic random = py.UseFile("random.py");
Console.WriteLine("random.py loaded.");
```

```
' Set the current directory to the IronPython libraries.
System.IO.Directory.SetCurrentDirectory(
    Environment.GetFolderPath(Environment.SpecialFolder.ProgramFiles) &
    "\IronPython 2.7\Lib")

' Create an instance of the random.py IronPython library.
Console.WriteLine("Loading random.py")
Dim py = Python.CreateRuntime()
Dim random As Object = py.UseFile("random.py")
Console.WriteLine("random.py loaded.")
```

9. After the code to load the `random.py` module, add the following code to create an array of integers. The array is passed to the `shuffle` method of the `random.py` module, which randomly sorts the values in the array.

```
// Initialize an enumerable set of integers.
int[] items = Enumerable.Range(1, 7).ToArray();

// Randomly shuffle the array of integers by using IronPython.
for (int i = 0; i < 5; i++)
{
    random.shuffle(items);
    foreach (int item in items)
    {
        Console.WriteLine(item);
    }
    Console.WriteLine("-----");
}
```

```
' Initialize an enumerable set of integers.
Dim items = Enumerable.Range(1, 7).ToArray()

' Randomly shuffle the array of integers by using IronPython.
For i = 0 To 4
    random.shuffle(items)
    For Each item In items
        Console.WriteLine(item)
    Next
    Console.WriteLine("-----")
Next
```

10. Save the file and press `Ctrl+F5` to build and run the application.

See also

- [System.Dynamic](#)
- [System.Dynamic.DynamicObject](#)
- [Using Type dynamic](#)
- [Early and Late Binding](#)
- [dynamic](#)
- [Implementing Dynamic Interfaces \(downloadable PDF from Microsoft TechNet\)](#)

Versioning with the Override and New Keywords (C# Programming Guide)

12/28/2021 • 5 minutes to read • [Edit Online](#)

The C# language is designed so that versioning between [base](#) and derived classes in different libraries can evolve and maintain backward compatibility. This means, for example, that the introduction of a new member in a base [class](#) with the same name as a member in a derived class is completely supported by C# and does not lead to unexpected behavior. It also means that a class must explicitly state whether a method is intended to override an inherited method, or whether a method is a new method that hides a similarly named inherited method.

In C#, derived classes can contain methods with the same name as base class methods.

- If the method in the derived class is not preceded by [new](#) or [override](#) keywords, the compiler will issue a warning and the method will behave as if the `new` keyword were present.
- If the method in the derived class is preceded with the `new` keyword, the method is defined as being independent of the method in the base class.
- If the method in the derived class is preceded with the `override` keyword, objects of the derived class will call that method instead of the base class method.
- In order to apply the `override` keyword to the method in the derived class, the base class method must be defined [virtual](#).
- The base class method can be called from within the derived class using the `base` keyword.
- The `override`, `virtual`, and `new` keywords can also be applied to properties, indexers, and events.

By default, C# methods are not virtual. If a method is declared as virtual, any class inheriting the method can implement its own version. To make a method virtual, the `virtual` modifier is used in the method declaration of the base class. The derived class can then override the base virtual method by using the `override` keyword or hide the virtual method in the base class by using the `new` keyword. If neither the `override` keyword nor the `new` keyword is specified, the compiler will issue a warning and the method in the derived class will hide the method in the base class.

To demonstrate this in practice, assume for a moment that Company A has created a class named `GraphicsClass`, which your program uses. The following is `GraphicsClass`:

```
class GraphicsClass
{
    public virtual void DrawLine() { }
    public virtual void DrawPoint() { }
}
```

Your company uses this class, and you use it to derive your own class, adding a new method:

```
class YourDerivedGraphicsClass : GraphicsClass
{
    public void DrawRectangle() { }
}
```

Your application is used without problems, until Company A releases a new version of `GraphicsClass`, which resembles the following code:

```
class GraphicsClass
{
    public virtual void DrawLine() { }
    public virtual void DrawPoint() { }
    public virtual void DrawRectangle() { }
}
```

The new version of `GraphicsClass` now contains a method named `DrawRectangle`. Initially, nothing occurs. The new version is still binary compatible with the old version. Any software that you have deployed will continue to work, even if the new class is installed on those computer systems. Any existing calls to the method `DrawRectangle` will continue to reference your version, in your derived class.

However, as soon as you recompile your application by using the new version of `GraphicsClass`, you will receive a warning from the compiler, CS0108. This warning informs you that you have to consider how you want your `DrawRectangle` method to behave in your application.

If you want your method to override the new base class method, use the `override` keyword:

```
class YourDerivedGraphicsClass : GraphicsClass
{
    public override void DrawRectangle() { }
}
```

The `override` keyword makes sure that any objects derived from `YourDerivedGraphicsClass` will use the derived class version of `DrawRectangle`. Objects derived from `YourDerivedGraphicsClass` can still access the base class version of `DrawRectangle` by using the base keyword:

```
base.DrawRectangle();
```

If you do not want your method to override the new base class method, the following considerations apply. To avoid confusion between the two methods, you can rename your method. This can be time-consuming and error-prone, and just not practical in some cases. However, if your project is relatively small, you can use Visual Studio's Refactoring options to rename the method. For more information, see [Refactoring Classes and Types \(Class Designer\)](#).

Alternatively, you can prevent the warning by using the keyword `new` in your derived class definition:

```
class YourDerivedGraphicsClass : GraphicsClass
{
    public new void DrawRectangle() { }
}
```

Using the `new` keyword tells the compiler that your definition hides the definition that is contained in the base class. This is the default behavior.

Override and Method Selection

When a method is named on a class, the C# compiler selects the best method to call if more than one method is compatible with the call, such as when there are two methods with the same name, and parameters that are compatible with the parameter passed. The following methods would be compatible:

```
public class Derived : Base
{
    public override void DoWork(int param) { }
    public void DoWork(double param) { }
}
```

When `DoWork` is called on an instance of `Derived`, the C# compiler will first try to make the call compatible with the versions of `DoWork` declared originally on `Derived`. Override methods are not considered as declared on a class, they are new implementations of a method declared on a base class. Only if the C# compiler cannot match the method call to an original method on `Derived`, it will try to match the call to an overridden method with the same name and compatible parameters. For example:

```
int val = 5;
Derived d = new Derived();
d.DoWork(val); // Calls DoWork(double).
```

Because the variable `val` can be converted to a double implicitly, the C# compiler calls `DoWork(double)` instead of `DoWork(int)`. There are two ways to avoid this. First, avoid declaring new methods with the same name as virtual methods. Second, you can instruct the C# compiler to call the virtual method by making it search the base class method list by casting the instance of `Derived` to `Base`. Because the method is virtual, the implementation of `DoWork(int)` on `Derived` will be called. For example:

```
((Base)d).DoWork(val); // Calls DoWork(int) on Derived.
```

For more examples of `new` and `override`, see [Knowing When to Use Override and New Keywords](#).

See also

- [C# Programming Guide](#)
- [The C# type system](#)
- [Methods](#)
- [Inheritance](#)

Knowing When to Use Override and New Keywords (C# Programming Guide)

12/28/2021 • 10 minutes to read • [Edit Online](#)

In C#, a method in a derived class can have the same name as a method in the base class. You can specify how the methods interact by using the `new` and `override` keywords. The `override` modifier *extends* the base class `virtual` method, and the `new` modifier *hides* an accessible base class method. The difference is illustrated in the examples in this topic.

In a console application, declare the following two classes, `BaseClass` and `DerivedClass`. `DerivedClass` inherits from `BaseClass`.

```
class BaseClass
{
    public void Method1()
    {
        Console.WriteLine("Base - Method1");
    }
}

class DerivedClass : BaseClass
{
    public void Method2()
    {
        Console.WriteLine("Derived - Method2");
    }
}
```

In the `Main` method, declare variables `bc`, `dc`, and `bcdc`.

- `bc` is of type `BaseClass`, and its value is of type `BaseClass`.
- `dc` is of type `DerivedClass`, and its value is of type `DerivedClass`.
- `bcdc` is of type `BaseClass`, and its value is of type `DerivedClass`. This is the variable to pay attention to.

Because `bc` and `bcdc` have type `BaseClass`, they can only directly access `Method1`, unless you use casting. Variable `dc` can access both `Method1` and `Method2`. These relationships are shown in the following code.

```

class Program
{
    static void Main(string[] args)
    {
        BaseClass bc = new BaseClass();
        DerivedClass dc = new DerivedClass();
        BaseClass bcdc = new DerivedClass();

        bc.Method1();
        dc.Method1();
        dc.Method2();
        bcdc.Method1();
    }
    // Output:
    // Base - Method1
    // Base - Method1
    // Derived - Method2
    // Base - Method1
}

```

Next, add the following `Method2` method to `BaseClass`. The signature of this method matches the signature of the `Method2` method in `DerivedClass`.

```

public void Method2()
{
    Console.WriteLine("Base - Method2");
}

```

Because `BaseClass` now has a `Method2` method, a second calling statement can be added for `BaseClass` variables `bc` and `bcdc`, as shown in the following code.

```

bc.Method1();
bc.Method2();
dc.Method1();
dc.Method2();
bcdc.Method1();
bcdc.Method2();

```

When you build the project, you see that the addition of the `Method2` method in `BaseClass` causes a warning. The warning says that the `Method2` method in `DerivedClass` hides the `Method2` method in `BaseClass`. You are advised to use the `new` keyword in the `Method2` definition if you intend to cause that result. Alternatively, you could rename one of the `Method2` methods to resolve the warning, but that is not always practical.

Before adding `new`, run the program to see the output produced by the additional calling statements. The following results are displayed.

```

// Output:
// Base - Method1
// Base - Method2
// Base - Method1
// Derived - Method2
// Base - Method1
// Base - Method2

```

The `new` keyword preserves the relationships that produce that output, but it suppresses the warning. The variables that have type `BaseClass` continue to access the members of `BaseClass`, and the variable that has type `DerivedClass` continues to access members in `DerivedClass` first, and then to consider members inherited

from `BaseClass` .

To suppress the warning, add the `new` modifier to the definition of `Method2` in `DerivedClass` , as shown in the following code. The modifier can be added before or after `public` .

```
public new void Method2()
{
    Console.WriteLine("Derived - Method2");
}
```

Run the program again to verify that the output has not changed. Also verify that the warning no longer appears. By using `new` , you are asserting that you are aware that the member that it modifies hides a member that is inherited from the base class. For more information about name hiding through inheritance, see [new Modifier](#).

To contrast this behavior to the effects of using `override` , add the following method to `DerivedClass` . The `override` modifier can be added before or after `public` .

```
public override void Method1()
{
    Console.WriteLine("Derived - Method1");
}
```

Add the `virtual` modifier to the definition of `Method1` in `BaseClass` . The `virtual` modifier can be added before or after `public` .

```
public virtual void Method1()
{
    Console.WriteLine("Base - Method1");
}
```

Run the project again. Notice especially the last two lines of the following output.

```
// Output:
// Base - Method1
// Base - Method2
// Derived - Method1
// Derived - Method2
// Derived - Method1
// Base - Method2
```

The use of the `override` modifier enables `bcd` to access the `Method1` method that is defined in `DerivedClass` . Typically, that is the desired behavior in inheritance hierarchies. You want objects that have values that are created from the derived class to use the methods that are defined in the derived class. You achieve that behavior by using `override` to extend the base class method.

The following code contains the full example.


```

using System;
using System.Text;

namespace OverrideAndNew
{
    class Program
    {
        static void Main(string[] args)
        {
            BaseClass bc = new BaseClass();
            DerivedClass dc = new DerivedClass();
            BaseClass bcdc = new DerivedClass();

            // The following two calls do what you would expect. They call
            // the methods that are defined in BaseClass.
            bc.Method1();
            bc.Method2();
            // Output:
            // Base - Method1
            // Base - Method2

            // The following two calls do what you would expect. They call
            // the methods that are defined in DerivedClass.
            dc.Method1();
            dc.Method2();
            // Output:
            // Derived - Method1
            // Derived - Method2

            // The following two calls produce different results, depending
            // on whether override (Method1) or new (Method2) is used.
            bcdc.Method1();
            bcdc.Method2();
            // Output:
            // Derived - Method1
            // Base - Method2
        }
    }

    class BaseClass
    {
        public virtual void Method1()
        {
            Console.WriteLine("Base - Method1");
        }

        public virtual void Method2()
        {
            Console.WriteLine("Base - Method2");
        }
    }

    class DerivedClass : BaseClass
    {
        public override void Method1()
        {
            Console.WriteLine("Derived - Method1");
        }

        public new void Method2()
        {
            Console.WriteLine("Derived - Method2");
        }
    }
}

```

The following example illustrates similar behavior in a different context. The example defines three classes: a base class named `Car` and two classes that are derived from it, `ConvertibleCar` and `Minivan`. The base class contains a `DescribeCar` method. The method displays a basic description of a car, and then calls `ShowDetails` to provide additional information. Each of the three classes defines a `ShowDetails` method. The `new` modifier is used to define `ShowDetails` in the `ConvertibleCar` class. The `override` modifier is used to define `ShowDetails` in the `Minivan` class.

```
// Define the base class, Car. The class defines two methods,  
// DescribeCar and ShowDetails. DescribeCar calls ShowDetails, and each derived  
// class also defines a ShowDetails method. The example tests which version of  
// ShowDetails is selected, the base class method or the derived class method.  
class Car  
{  
    public void DescribeCar()  
    {  
        System.Console.WriteLine("Four wheels and an engine.");  
        ShowDetails();  
    }  
  
    public virtual void ShowDetails()  
    {  
        System.Console.WriteLine("Standard transportation.");  
    }  
}  
  
// Define the derived classes.  
  
// Class ConvertibleCar uses the new modifier to acknowledge that ShowDetails  
// hides the base class method.  
class ConvertibleCar : Car  
{  
    public new void ShowDetails()  
    {  
        System.Console.WriteLine("A roof that opens up.");  
    }  
}  
  
// Class Minivan uses the override modifier to specify that ShowDetails  
// extends the base class method.  
class Minivan : Car  
{  
    public override void ShowDetails()  
    {  
        System.Console.WriteLine("Carries seven people.");  
    }  
}
```

The example tests which version of `ShowDetails` is called. The following method, `TestCars1`, declares an instance of each class, and then calls `DescribeCar` on each instance.

```

public static void TestCars1()
{
    System.Console.WriteLine("\nTestCars1");
    System.Console.WriteLine("-----");

    Car car1 = new Car();
    car1.DescribeCar();
    System.Console.WriteLine("-----");

    // Notice the output from this test case. The new modifier is
    // used in the definition of ShowDetails in the ConvertibleCar
    // class.

    ConvertibleCar car2 = new ConvertibleCar();
    car2.DescribeCar();
    System.Console.WriteLine("-----");

    Minivan car3 = new Minivan();
    car3.DescribeCar();
    System.Console.WriteLine("-----");
}

```

`TestCars1` produces the following output. Notice especially the results for `car2`, which probably are not what you expected. The type of the object is `ConvertibleCar`, but `DescribeCar` does not access the version of `ShowDetails` that is defined in the `ConvertibleCar` class because that method is declared with the `new` modifier, not the `override` modifier. As a result, a `ConvertibleCar` object displays the same description as a `Car` object. Contrast the results for `car3`, which is a `Minivan` object. In this case, the `ShowDetails` method that is declared in the `Minivan` class overrides the `ShowDetails` method that is declared in the `Car` class, and the description that is displayed describes a minivan.

```

// TestCars1
// -----
// Four wheels and an engine.
// Standard transportation.
// -----
// Four wheels and an engine.
// Standard transportation.
// -----
// Four wheels and an engine.
// Carries seven people.
// -----

```

`TestCars2` creates a list of objects that have type `Car`. The values of the objects are instantiated from the `Car`, `ConvertibleCar`, and `Minivan` classes. `DescribeCar` is called on each element of the list. The following code shows the definition of `TestCars2`.

```

public static void TestCars2()
{
    System.Console.WriteLine("\nTestCars2");
    System.Console.WriteLine("-----");

    var cars = new List<Car> { new Car(), new ConvertibleCar(),
                              new Minivan() };

    foreach (var car in cars)
    {
        car.DescribeCar();
        System.Console.WriteLine("-----");
    }
}

```

The following output is displayed. Notice that it is the same as the output that is displayed by `TestCars1`. The `ShowDetails` method of the `ConvertibleCar` class is not called, regardless of whether the type of the object is `ConvertibleCar`, as in `TestCars1`, or `Car`, as in `TestCars2`. Conversely, `car3` calls the `ShowDetails` method from the `Minivan` class in both cases, whether it has type `Minivan` or type `Car`.

```
// TestCars2
// -----
// Four wheels and an engine.
// Standard transportation.
// -----
// Four wheels and an engine.
// Standard transportation.
// -----
// Four wheels and an engine.
// Carries seven people.
// -----
```

Methods `TestCars3` and `TestCars4` complete the example. These methods call `ShowDetails` directly, first from objects declared to have type `ConvertibleCar` and `Minivan` (`TestCars3`), then from objects declared to have type `Car` (`TestCars4`). The following code defines these two methods.

```
public static void TestCars3()
{
    System.Console.WriteLine("\nTestCars3");
    System.Console.WriteLine("-----");
    ConvertibleCar car2 = new ConvertibleCar();
    Minivan car3 = new Minivan();
    car2.ShowDetails();
    car3.ShowDetails();
}

public static void TestCars4()
{
    System.Console.WriteLine("\nTestCars4");
    System.Console.WriteLine("-----");
    Car car2 = new ConvertibleCar();
    Car car3 = new Minivan();
    car2.ShowDetails();
    car3.ShowDetails();
}
```

The methods produce the following output, which corresponds to the results from the first example in this topic.

```
// TestCars3
// -----
// A roof that opens up.
// Carries seven people.

// TestCars4
// -----
// Standard transportation.
// Carries seven people.
```

The following code shows the complete project and its output.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
```

```

namespace overrideAndnew2
{
    class Program
    {
        static void Main(string[] args)
        {
            // Declare objects of the derived classes and test which version
            // of ShowDetails is run, base or derived.
            TestCars1();

            // Declare objects of the base class, instantiated with the
            // derived classes, and repeat the tests.
            TestCars2();

            // Declare objects of the derived classes and call ShowDetails
            // directly.
            TestCars3();

            // Declare objects of the base class, instantiated with the
            // derived classes, and repeat the tests.
            TestCars4();
        }

        public static void TestCars1()
        {
            System.Console.WriteLine("\nTestCars1");
            System.Console.WriteLine("-----");

            Car car1 = new Car();
            car1.DescribeCar();
            System.Console.WriteLine("-----");

            // Notice the output from this test case. The new modifier is
            // used in the definition of ShowDetails in the ConvertibleCar
            // class.
            ConvertibleCar car2 = new ConvertibleCar();
            car2.DescribeCar();
            System.Console.WriteLine("-----");

            Minivan car3 = new Minivan();
            car3.DescribeCar();
            System.Console.WriteLine("-----");
        }

        // Output:
        // TestCars1
        // -----
        // Four wheels and an engine.
        // Standard transportation.
        // -----
        // Four wheels and an engine.
        // Standard transportation.
        // -----
        // Four wheels and an engine.
        // Carries seven people.
        // -----

        public static void TestCars2()
        {
            System.Console.WriteLine("\nTestCars2");
            System.Console.WriteLine("-----");

            var cars = new List<Car> { new Car(), new ConvertibleCar(),
                                      new Minivan() };

            foreach (var car in cars)
            {
                car.DescribeCar();
                System.Console.WriteLine("-----");
            }
        }
    }
}

```

```

    }
    // Output:
    // TestCars2
    // -----
    // Four wheels and an engine.
    // Standard transportation.
    // -----
    // Four wheels and an engine.
    // Standard transportation.
    // -----
    // Four wheels and an engine.
    // Carries seven people.
    // -----

    public static void TestCars3()
    {
        System.Console.WriteLine("\nTestCars3");
        System.Console.WriteLine("-----");
        ConvertibleCar car2 = new ConvertibleCar();
        Minivan car3 = new Minivan();
        car2.ShowDetails();
        car3.ShowDetails();
    }
    // Output:
    // TestCars3
    // -----
    // A roof that opens up.
    // Carries seven people.

    public static void TestCars4()
    {
        System.Console.WriteLine("\nTestCars4");
        System.Console.WriteLine("-----");
        Car car2 = new ConvertibleCar();
        Car car3 = new Minivan();
        car2.ShowDetails();
        car3.ShowDetails();
    }
    // Output:
    // TestCars4
    // -----
    // Standard transportation.
    // Carries seven people.
}

// Define the base class, Car. The class defines two virtual methods,
// DescribeCar and ShowDetails. DescribeCar calls ShowDetails, and each derived
// class also defines a ShowDetails method. The example tests which version of
// ShowDetails is used, the base class method or the derived class method.
class Car
{
    public virtual void DescribeCar()
    {
        System.Console.WriteLine("Four wheels and an engine.");
        ShowDetails();
    }

    public virtual void ShowDetails()
    {
        System.Console.WriteLine("Standard transportation.");
    }
}

// Define the derived classes.

// Class ConvertibleCar uses the new modifier to acknowledge that ShowDetails
// hides the base class method.
class ConvertibleCar : Car
{

```

```
        public new void ShowDetails()
        {
            System.Console.WriteLine("A roof that opens up.");
        }
    }

    // Class Minivan uses the override modifier to specify that ShowDetails
    // extends the base class method.
    class Minivan : Car
    {
        public override void ShowDetails()
        {
            System.Console.WriteLine("Carries seven people.");
        }
    }
}
```

See also

- [C# Programming Guide](#)
- [The C# type system](#)
- [Versioning with the Override and New Keywords](#)
- [base](#)
- [abstract](#)

How to override the ToString method (C# Programming Guide)

12/28/2021 • 2 minutes to read • [Edit Online](#)

Every class or struct in C# implicitly inherits the [Object](#) class. Therefore, every object in C# gets the [ToString](#) method, which returns a string representation of that object. For example, all variables of type `int` have a `ToString` method, which enables them to return their contents as a string:

```
int x = 42;
string strx = x.ToString();
Console.WriteLine(strx);
// Output:
// 42
```

When you create a custom class or struct, you should override the [ToString](#) method in order to provide information about your type to client code.

For information about how to use format strings and other types of custom formatting with the `ToString` method, see [Formatting Types](#).

IMPORTANT

When you decide what information to provide through this method, consider whether your class or struct will ever be used by untrusted code. Be careful to ensure that you do not provide any information that could be exploited by malicious code.

To override the `ToString` method in your class or struct:

1. Declare a `ToString` method with the following modifiers and return type:

```
public override string ToString(){} 
```

2. Implement the method so that it returns a string.

The following example returns the name of the class in addition to the data specific to a particular instance of the class.

```
class Person
{
    public string Name { get; set; }
    public int Age { get; set; }

    public override string ToString()
    {
        return "Person: " + Name + " " + Age;
    }
}
```

You can test the `ToString` method as shown in the following code example:


```
Person person = new Person { Name = "John", Age = 12 };  
Console.WriteLine(person);  
// Output:  
// Person: John 12
```

See also

- [IFormattable](#)
- [C# Programming Guide](#)
- [The C# type system](#)
- [Strings](#)
- [string](#)
- [override](#)
- [virtual](#)
- [Formatting Types](#)

Members (C# Programming Guide)

12/28/2021 • 2 minutes to read • [Edit Online](#)

Classes and structs have members that represent their data and behavior. A class's members include all the members declared in the class, along with all members (except constructors and finalizers) declared in all classes in its inheritance hierarchy. Private members in base classes are inherited but are not accessible from derived classes.

The following table lists the kinds of members a class or struct may contain:

MEMBER	DESCRIPTION
Fields	Fields are variables declared at class scope. A field may be a built-in numeric type or an instance of another class. For example, a calendar class may have a field that contains the current date.
Constants	Constants are fields whose value is set at compile time and cannot be changed.
Properties	Properties are methods on a class that are accessed as if they were fields on that class. A property can provide protection for a class field to keep it from being changed without the knowledge of the object.
Methods	Methods define the actions that a class can perform. Methods can take parameters that provide input data, and can return output data through parameters. Methods can also return a value directly, without using a parameter.
Events	Events provide notifications about occurrences, such as button clicks or the successful completion of a method, to other objects. Events are defined and triggered by using delegates.
Operators	Overloaded operators are considered type members. When you overload an operator, you define it as a public static method in a type. For more information, see Operator overloading .
Indexers	Indexers enable an object to be indexed in a manner similar to arrays.
Constructors	Constructors are methods that are called when the object is first created. They are often used to initialize the data of an object.
Finalizers	Finalizers are used very rarely in C#. They are methods that are called by the runtime execution engine when the object is about to be removed from memory. They are generally used to make sure that any resources which must be released are handled appropriately.

MEMBER	DESCRIPTION
Nested Types	Nested types are types declared within another type. Nested types are often used to describe objects that are used only by the types that contain them.

See also

- [C# Programming Guide](#)
- [Classes](#)

Abstract and Sealed Classes and Class Members (C# Programming Guide)

12/28/2021 • 2 minutes to read • [Edit Online](#)

The [abstract](#) keyword enables you to create classes and [class](#) members that are incomplete and must be implemented in a derived class.

The [sealed](#) keyword enables you to prevent the inheritance of a class or certain class members that were previously marked [virtual](#).

Abstract Classes and Class Members

Classes can be declared as abstract by putting the keyword `abstract` before the class definition. For example:

```
public abstract class A
{
    // Class members here.
}
```

An abstract class cannot be instantiated. The purpose of an abstract class is to provide a common definition of a base class that multiple derived classes can share. For example, a class library may define an abstract class that is used as a parameter to many of its functions, and require programmers using that library to provide their own implementation of the class by creating a derived class.

Abstract classes may also define abstract methods. This is accomplished by adding the keyword `abstract` before the return type of the method. For example:

```
public abstract class A
{
    public abstract void DoWork(int i);
}
```

Abstract methods have no implementation, so the method definition is followed by a semicolon instead of a normal method block. Derived classes of the abstract class must implement all abstract methods. When an abstract class inherits a virtual method from a base class, the abstract class can override the virtual method with an abstract method. For example:

```
// compile with: -target:library
public class D
{
    public virtual void DoWork(int i)
    {
        // Original implementation.
    }
}

public abstract class E : D
{
    public abstract override void DoWork(int i);
}

public class F : E
{
    public override void DoWork(int i)
    {
        // New implementation.
    }
}
```

If a `virtual` method is declared `abstract`, it is still virtual to any class inheriting from the abstract class. A class inheriting an abstract method cannot access the original implementation of the method—in the previous example, `DoWork` on class `F` cannot call `DoWork` on class `D`. In this way, an abstract class can force derived classes to provide new method implementations for virtual methods.

Sealed Classes and Class Members

Classes can be declared as `sealed` by putting the keyword `sealed` before the class definition. For example:

```
public sealed class D
{
    // Class members here.
}
```

A sealed class cannot be used as a base class. For this reason, it cannot also be an abstract class. Sealed classes prevent derivation. Because they can never be used as a base class, some run-time optimizations can make calling sealed class members slightly faster.

A method, indexer, property, or event, on a derived class that is overriding a virtual member of the base class can declare that member as sealed. This negates the virtual aspect of the member for any further derived class. This is accomplished by putting the `sealed` keyword before the `override` keyword in the class member declaration. For example:

```
public class D : C
{
    public sealed override void DoWork() { }
```

See also

- [C# Programming Guide](#)
- [The C# type system](#)
- [Inheritance](#)
- [Methods](#)

- [Fields](#)
- [How to define abstract properties](#)

Static Classes and Static Class Members (C# Programming Guide)

12/28/2021 • 5 minutes to read • [Edit Online](#)

A **static** class is basically the same as a non-static class, but there is one difference: a static class cannot be instantiated. In other words, you cannot use the **new** operator to create a variable of the class type. Because there is no instance variable, you access the members of a static class by using the class name itself. For example, if you have a static class that is named `UtilityClass` that has a public static method named `MethodA`, you call the method as shown in the following example:

```
UtilityClass.MethodA();
```

A static class can be used as a convenient container for sets of methods that just operate on input parameters and do not have to get or set any internal instance fields. For example, in the .NET Class Library, the static **System.Math** class contains methods that perform mathematical operations, without any requirement to store or retrieve data that is unique to a particular instance of the **Math** class. That is, you apply the members of the class by specifying the class name and the method name, as shown in the following example.

```
double dub = -3.14;
Console.WriteLine(Math.Abs(dub));
Console.WriteLine(Math.Floor(dub));
Console.WriteLine(Math.Round(Math.Abs(dub)));

// Output:
// 3.14
// -4
// 3
```

As is the case with all class types, the type information for a static class is loaded by the .NET runtime when the program that references the class is loaded. The program cannot specify exactly when the class is loaded. However, it is guaranteed to be loaded and to have its fields initialized and its static constructor called before the class is referenced for the first time in your program. A static constructor is only called one time, and a static class remains in memory for the lifetime of the application domain in which your program resides.

NOTE

To create a non-static class that allows only one instance of itself to be created, see [Implementing Singleton in C#](#).

The following list provides the main features of a static class:

- Contains only static members.
- Cannot be instantiated.
- Is sealed.
- Cannot contain [Instance Constructors](#).

Creating a static class is therefore basically the same as creating a class that contains only static members and a private constructor. A private constructor prevents the class from being instantiated. The advantage of using a static class is that the compiler can check to make sure that no instance members are accidentally added. The

compiler will guarantee that instances of this class cannot be created.

Static classes are sealed and therefore cannot be inherited. They cannot inherit from any class except [Object](#). Static classes cannot contain an instance constructor. However, they can contain a static constructor. Non-static classes should also define a static constructor if the class contains static members that require non-trivial initialization. For more information, see [Static Constructors](#).

Example

Here is an example of a static class that contains two methods that convert temperature from Celsius to Fahrenheit and from Fahrenheit to Celsius:

```
public static class TemperatureConverter
{
    public static double CelsiusToFahrenheit(string temperatureCelsius)
    {
        // Convert argument to double for calculations.
        double celsius = Double.Parse(temperatureCelsius);

        // Convert Celsius to Fahrenheit.
        double fahrenheit = (celsius * 9 / 5) + 32;

        return fahrenheit;
    }

    public static double FahrenheitToCelsius(string temperatureFahrenheit)
    {
        // Convert argument to double for calculations.
        double fahrenheit = Double.Parse(temperatureFahrenheit);

        // Convert Fahrenheit to Celsius.
        double celsius = (fahrenheit - 32) * 5 / 9;

        return celsius;
    }
}

class TestTemperatureConverter
{
    static void Main()
    {
        Console.WriteLine("Please select the convertor direction");
        Console.WriteLine("1. From Celsius to Fahrenheit.");
        Console.WriteLine("2. From Fahrenheit to Celsius.");
        Console.Write(":");

        string selection = Console.ReadLine();
        double F, C = 0;

        switch (selection)
        {
            case "1":
                Console.Write("Please enter the Celsius temperature: ");
                F = TemperatureConverter.CelsiusToFahrenheit(Console.ReadLine());
                Console.WriteLine("Temperature in Fahrenheit: {0:F2}", F);
                break;

            case "2":
                Console.Write("Please enter the Fahrenheit temperature: ");
                C = TemperatureConverter.FahrenheitToCelsius(Console.ReadLine());
                Console.WriteLine("Temperature in Celsius: {0:F2}", C);
                break;

            default:
                Console.WriteLine("Please select a convertor.");
                break;
        }
    }
}
```



```

        break;
    }

    // Keep the console window open in debug mode.
    Console.WriteLine("Press any key to exit.");
    Console.ReadKey();
}
}
/* Example Output:
    Please select the convertor direction
    1. From Celsius to Fahrenheit.
    2. From Fahrenheit to Celsius.
    :2
    Please enter the Fahrenheit temperature: 20
    Temperature in Celsius: -6.67
    Press any key to exit.
*/

```

Static Members

A non-static class can contain static methods, fields, properties, or events. The static member is callable on a class even when no instance of the class has been created. The static member is always accessed by the class name, not the instance name. Only one copy of a static member exists, regardless of how many instances of the class are created. Static methods and properties cannot access non-static fields and events in their containing type, and they cannot access an instance variable of any object unless it's explicitly passed in a method parameter.

It is more typical to declare a non-static class with some static members, than to declare an entire class as static. Two common uses of static fields are to keep a count of the number of objects that have been instantiated, or to store a value that must be shared among all instances.

Static methods can be overloaded but not overridden, because they belong to the class, and not to any instance of the class.

Although a field cannot be declared as `static const`, a `const` field is essentially static in its behavior. It belongs to the type, not to instances of the type. Therefore, `const` fields can be accessed by using the same `ClassName.MemberName` notation that's used for static fields. No object instance is required.

C# does not support static local variables (that is, variables that are declared in method scope).

You declare static class members by using the `static` keyword before the return type of the member, as shown in the following example:

```

public class Automobile
{
    public static int NumberOfWheels = 4;

    public static int SizeOfGasTank
    {
        get
        {
            return 15;
        }
    }

    public static void Drive() { }

    public static event EventType RunOutOfGas;

    // Other non-static fields and properties...
}

```

Static members are initialized before the static member is accessed for the first time and before the static constructor, if there is one, is called. To access a static class member, use the name of the class instead of a variable name to specify the location of the member, as shown in the following example:

```
Automobile.Drive();  
int i = Automobile.NumberOfWheels;
```

If your class contains static fields, provide a static constructor that initializes them when the class is loaded.

A call to a static method generates a call instruction in Microsoft intermediate language (MSIL), whereas a call to an instance method generates a `callvirt` instruction, which also checks for null object references. However, most of the time the performance difference between the two is not significant.

C# Language Specification

For more information, see [Static classes](#) and [Static and instance members](#) in the [C# Language Specification](#). The language specification is the definitive source for C# syntax and usage.

See also

- [C# Programming Guide](#)
- [static](#)
- [Classes](#)
- [class](#)
- [Static Constructors](#)
- [Instance Constructors](#)

Access Modifiers (C# Programming Guide)

12/28/2021 • 4 minutes to read • [Edit Online](#)

All types and type members have an accessibility level. The accessibility level controls whether they can be used from other code in your assembly or other assemblies. Use the following access modifiers to specify the accessibility of a type or member when you declare it:

- **public**: The type or member can be accessed by any other code in the same assembly or another assembly that references it. The accessibility level of public members of a type is controlled by the accessibility level of the type itself.
- **private**: The type or member can be accessed only by code in the same `class` or `struct`.
- **protected**: The type or member can be accessed only by code in the same `class`, or in a `class` that is derived from that `class`.
- **internal**: The type or member can be accessed by any code in the same assembly, but not from another assembly.
- **protected internal**: The type or member can be accessed by any code in the assembly in which it's declared, or from within a derived `class` in another assembly.
- **private protected**: The type or member can be accessed by types derived from the `class` that are declared within its containing assembly.

Summary table

CALLER'S LOCATION	<code>PUBLIC</code>	<code>PROTECTED INTERNAL</code>	<code>PROTECTED</code>	<code>INTERNAL</code>	<code>PRIVATE PROTECTED</code>	<code>PRIVATE</code>
Within the class	✓	✓	✓	✓	✓	✓
Derived class (same assembly)	✓	✓	✓	✓	✓	✗
Non-derived class (same assembly)	✓	✓	✗	✓	✗	✗
Derived class (different assembly)	✓	✓	✓	✗	✗	✗
Non-derived class (different assembly)	✓	✗	✗	✗	✗	✗

The following examples demonstrate how to specify access modifiers on a type and member:

```
public class Bicycle
{
    public void Pedal() { }
}
```

Not all access modifiers are valid for all types or members in all contexts. In some cases, the accessibility of a type member is constrained by the accessibility of its containing type.

Class, record, and struct accessibility

Classes, records, and structs declared directly within a namespace (in other words, that aren't nested within other classes or structs) can be either `public` or `internal`. `internal` is the default if no access modifier is specified.

Struct members, including nested classes and structs, can be declared `public`, `internal`, or `private`. Class members, including nested classes and structs, can be `public`, `protected internal`, `protected`, `internal`, `private protected`, or `private`. Class and struct members, including nested classes and structs, have `private` access by default. Private nested types aren't accessible from outside the containing type.

Derived classes and derived records can't have greater accessibility than their base types. You can't declare a public class `B` that derives from an internal class `A`. If allowed, it would have the effect of making `A` public, because all `protected` or `internal` members of `A` are accessible from the derived class.

You can enable specific other assemblies to access your internal types by using the `InternalsVisibleToAttribute`. For more information, see [Friend Assemblies](#).

Class, record, and struct member accessibility

Class and record members (including nested classes, records and structs) can be declared with any of the six types of access. Struct members can't be declared as `protected`, `protected internal`, or `private protected` because structs don't support inheritance.

Normally, the accessibility of a member isn't greater than the accessibility of the type that contains it. However, a `public` member of an internal class might be accessible from outside the assembly if the member implements interface methods or overrides virtual methods that are defined in a public base class.

The type of any member field, property, or event must be at least as accessible as the member itself. Similarly, the return type and the parameter types of any method, indexer, or delegate must be at least as accessible as the member itself. For example, you can't have a `public` method `M` that returns a class `C` unless `C` is also `public`. Likewise, you can't have a `protected` property of type `A` if `A` is declared as `private`.

User-defined operators must always be declared as `public` and `static`. For more information, see [Operator overloading](#).

Finalizers can't have accessibility modifiers.

To set the access level for a `class`, `record`, or `struct` member, add the appropriate keyword to the member declaration, as shown in the following example.

```
// public class:
public class Tricycle
{
    // protected method:
    protected void Pedal() { }

    // private field:
    private int wheels = 3;

    // protected internal property:
    protected internal int Wheels
    {
        get { return wheels; }
    }
}
```

Other types

Interfaces declared directly within a namespace can be `public` or `internal` and, just like classes and structs, interfaces default to `internal` access. Interface members are `public` by default because the purpose of an interface is to enable other types to access a class or struct. Interface member declarations may include any access modifier. This is most useful for static methods to provide common implementations needed by all implementors of a class.

Enumeration members are always `public`, and no access modifiers can be applied.

Delegates behave like classes and structs. By default, they have `internal` access when declared directly within a namespace, and `private` access when nested.

C# language specification

For more information, see the [C# Language Specification](#). The language specification is the definitive source for C# syntax and usage.

See also

- [C# Programming Guide](#)
- [The C# type system](#)
- [Interfaces](#)
- [private](#)
- [public](#)
- [internal](#)
- [protected](#)
- [protected internal](#)
- [private protected](#)
- [class](#)
- [struct](#)
- [interface](#)

Fields (C# Programming Guide)

12/28/2021 • 3 minutes to read • [Edit Online](#)

A *field* is a variable of any type that is declared directly in a [class](#) or [struct](#). Fields are *members* of their containing type.

A class or struct may have instance fields, static fields, or both. Instance fields are specific to an instance of a type. If you have a class T, with an instance field F, you can create two objects of type T, and modify the value of F in each object without affecting the value in the other object. By contrast, a static field belongs to the type itself, and is shared among all instances of that type. You can access the static field only by using the type name. If you access the static field by an instance name, you get [CS0176](#) compile-time error.

Generally, you should use fields only for variables that have private or protected accessibility. Data that your type exposes to client code should be provided through [methods](#), [properties](#), and [indexers](#). By using these constructs for indirect access to internal fields, you can guard against invalid input values. A private field that stores the data exposed by a public property is called a *backing store* or *backing field*.

Fields typically store the data that must be accessible to more than one type method and must be stored for longer than the lifetime of any single method. For example, a type that represents a calendar date might have three integer fields: one for the month, one for the day, and one for the year. Variables that are not used outside the scope of a single method should be declared as *local variables* within the method body itself.

Fields are declared in the class or struct block by specifying the access level of the field, followed by the type of the field, followed by the name of the field. For example:

```

public class CalendarEntry
{
    // private field (Located near wrapping "Date" property).
    private DateTime _date;

    // Public property exposes _date field safely.
    public DateTime Date
    {
        get
        {
            return _date;
        }
        set
        {
            // Set some reasonable boundaries for likely birth dates.
            if (value.Year > 1900 && value.Year <= DateTime.Today.Year)
            {
                _date = value;
            }
            else
            {
                throw new ArgumentOutOfRangeException();
            }
        }
    }

    // public field (Generally not recommended).
    public string Day;

    // Public method also exposes _date field safely.
    // Example call: birthday.SetDate("1975, 6, 30");
    public void SetDate(string dateString)
    {
        DateTime dt = Convert.ToDateTime(dateString);

        // Set some reasonable boundaries for likely birth dates.
        if (dt.Year > 1900 && dt.Year <= DateTime.Today.Year)
        {
            _date = dt;
        }
        else
        {
            throw new ArgumentOutOfRangeException();
        }
    }

    public TimeSpan GetTimeSpan(string dateString)
    {
        DateTime dt = Convert.ToDateTime(dateString);

        if (dt.Ticks < _date.Ticks)
        {
            return _date - dt;
        }
        else
        {
            throw new ArgumentOutOfRangeException();
        }
    }
}

```

To access a field in an instance, add a period after the instance name, followed by the name of the field, as in `instancename._fieldName`. For example:

```
CalendarEntry birthday = new CalendarEntry();  
birthday.Day = "Saturday";
```

A field can be given an initial value by using the assignment operator when the field is declared. To automatically assign the `Day` field to `"Monday"`, for example, you would declare `Day` as in the following example:

```
public class CalendarDateWithInitialization  
{  
    public string Day = "Monday";  
    //...  
}
```

Fields are initialized immediately before the constructor for the object instance is called. If the constructor assigns the value of a field, it will overwrite any value given during field declaration. For more information, see [Using Constructors](#).

NOTE

A field initializer cannot refer to other instance fields.

Fields can be marked as [public](#), [private](#), [protected](#), [internal](#), [protected internal](#), or [private protected](#). These access modifiers define how users of the type can access the fields. For more information, see [Access Modifiers](#).

A field can optionally be declared [static](#). This makes the field available to callers at any time, even if no instance of the type exists. For more information, see [Static Classes and Static Class Members](#).

A field can be declared [readonly](#). A read-only field can only be assigned a value during initialization or in a constructor. A `static readonly` field is very similar to a constant, except that the C# compiler does not have access to the value of a static read-only field at compile time, only at run time. For more information, see [Constants](#).

C# language specification

For more information, see the [C# Language Specification](#). The language specification is the definitive source for C# syntax and usage.

See also

- [C# Programming Guide](#)
- [The C# type system](#)
- [Using Constructors](#)
- [Inheritance](#)
- [Access Modifiers](#)
- [Abstract and Sealed Classes and Class Members](#)

Constants (C# Programming Guide)

12/28/2021 • 2 minutes to read • [Edit Online](#)

Constants are immutable values which are known at compile time and do not change for the life of the program. Constants are declared with the `const` modifier. Only the C# [built-in types](#) (excluding `System.Object`) may be declared as `const`. User-defined types, including classes, structs, and arrays, cannot be `const`. Use the `readonly` modifier to create a class, struct, or array that is initialized one time at run time (for example in a constructor) and thereafter cannot be changed.

C# does not support `const` methods, properties, or events.

The enum type enables you to define named constants for integral built-in types (for example `int`, `uint`, `long`, and so on). For more information, see [enum](#).

Constants must be initialized as they are declared. For example:

```
class Calendar1
{
    public const int Months = 12;
}
```

In this example, the constant `Months` is always 12, and it cannot be changed even by the class itself. In fact, when the compiler encounters a constant identifier in C# source code (for example, `Months`), it substitutes the literal value directly into the intermediate language (IL) code that it produces. Because there is no variable address associated with a constant at run time, `const` fields cannot be passed by reference and cannot appear as an l-value in an expression.

NOTE

Use caution when you refer to constant values defined in other code such as DLLs. If a new version of the DLL defines a new value for the constant, your program will still hold the old literal value until it is recompiled against the new version.

Multiple constants of the same type can be declared at the same time, for example:

```
class Calendar2
{
    public const int Months = 12, Weeks = 52, Days = 365;
}
```

The expression that is used to initialize a constant can refer to another constant if it does not create a circular reference. For example:

```
class Calendar3
{
    public const int Months = 12;
    public const int Weeks = 52;
    public const int Days = 365;

    public const double DaysPerWeek = (double) Days / (double) Weeks;
    public const double DaysPerMonth = (double) Days / (double) Months;
}
```

Constants can be marked as [public](#), [private](#), [protected](#), [internal](#), [protected internal](#) or [private protected](#). These access modifiers define how users of the class can access the constant. For more information, see [Access Modifiers](#).

Constants are accessed as if they were [static](#) fields because the value of the constant is the same for all instances of the type. You do not use the `static` keyword to declare them. Expressions that are not in the class that defines the constant must use the class name, a period, and the name of the constant to access the constant. For example:

```
int birthstones = Calendar.Months;
```

C# Language Specification

For more information, see the [C# Language Specification](#). The language specification is the definitive source for C# syntax and usage.

See also

- [C# Programming Guide](#)
- [Properties](#)
- [Types](#)
- [readonly](#)
- [Immutability in C# Part One: Kinds of Immutability](#)

How to define abstract properties (C# Programming Guide)

12/28/2021 • 2 minutes to read • [Edit Online](#)

The following example shows how to define [abstract](#) properties. An abstract property declaration does not provide an implementation of the property accessors -- it declares that the class supports properties, but leaves the accessor implementation to derived classes. The following example demonstrates how to implement the abstract properties inherited from a base class.

This sample consists of three files, each of which is compiled individually and its resulting assembly is referenced by the next compilation:

- abstractshape.cs: the `Shape` class that contains an abstract `Area` property.
- shapes.cs: The subclasses of the `Shape` class.
- shapetest.cs: A test program to display the areas of some `Shape`-derived objects.

To compile the example, use the following command:

```
csc abstractshape.cs shapes.cs shapetest.cs
```

This will create the executable file shapetest.exe.

Examples

This file declares the `Shape` class that contains the `Area` property of the type `double`.

```
// compile with: csc -target:library abstractshape.cs
public abstract class Shape
{
    private string name;

    public Shape(string s)
    {
        // calling the set accessor of the Id property.
        Id = s;
    }

    public string Id
    {
        get
        {
            return name;
        }

        set
        {
            name = value;
        }
    }

    // Area is a read-only property - only a get accessor is needed:
    public abstract double Area
    {
        get;
    }

    public override string ToString()
    {
        return $"{Id} Area = {Area:F2}";
    }
}
```

- Modifiers on the property are placed on the property declaration itself. For example:

```
public abstract double Area
```

- When declaring an abstract property (such as `Area` in this example), you simply indicate what property accessors are available, but do not implement them. In this example, only a `get` accessor is available, so the property is read-only.

The following code shows three subclasses of `Shape` and how they override the `Area` property to provide their own implementation.

```
// compile with: csc -target:library -reference:abstractshape.dll shapes.cs
public class Square : Shape
{
    private int side;

    public Square(int side, string id)
        : base(id)
    {
        this.side = side;
    }

    public override double Area
    {
        get
        {
            // Given the side, return the area of a square:
            return side * side;
        }
    }
}

public class Circle : Shape
{
    private int radius;

    public Circle(int radius, string id)
        : base(id)
    {
        this.radius = radius;
    }

    public override double Area
    {
        get
        {
            // Given the radius, return the area of a circle:
            return radius * radius * System.Math.PI;
        }
    }
}

public class Rectangle : Shape
{
    private int width;
    private int height;

    public Rectangle(int width, int height, string id)
        : base(id)
    {
        this.width = width;
        this.height = height;
    }

    public override double Area
    {
        get
        {
            // Given the width and height, return the area of a rectangle:
            return width * height;
        }
    }
}
```

The following code shows a test program that creates a number of `Shape`-derived objects and prints out their areas.

```
// compile with: csc -reference:abstractshape.dll;shapes.dll shapetest.cs
class TestClass
{
    static void Main()
    {
        Shape[] shapes =
        {
            new Square(5, "Square #1"),
            new Circle(3, "Circle #1"),
            new Rectangle( 4, 5, "Rectangle #1")
        };

        System.Console.WriteLine("Shapes Collection");
        foreach (Shape s in shapes)
        {
            System.Console.WriteLine(s);
        }
    }
}
/* Output:
    Shapes Collection
    Square #1 Area = 25.00
    Circle #1 Area = 28.27
    Rectangle #1 Area = 20.00
*/
```

See also

- [C# Programming Guide](#)
- [The C# type system](#)
- [Abstract and Sealed Classes and Class Members](#)
- [Properties](#)

How to define constants in C#

12/28/2021 • 2 minutes to read • [Edit Online](#)

Constants are fields whose values are set at compile time and can never be changed. Use constants to provide meaningful names instead of numeric literals ("magic numbers") for special values.

NOTE

In C# the `#define` preprocessor directive cannot be used to define constants in the way that is typically used in C and C++.

To define constant values of integral types (`int`, `byte`, and so on) use an enumerated type. For more information, see [enum](#).

To define non-integral constants, one approach is to group them in a single static class named `Constants`. This will require that all references to the constants be prefaced with the class name, as shown in the following example.

Example

```
using System;

static class Constants
{
    public const double Pi = 3.14159;
    public const int SpeedOfLight = 300000; // km per sec.
}

class Program
{
    static void Main()
    {
        double radius = 5.3;
        double area = Constants.Pi * (radius * radius);
        int secsFromSun = 149476000 / Constants.SpeedOfLight; // in km
        Console.WriteLine(secsFromSun);
    }
}
```

The use of the class name qualifier helps ensure that you and others who use the constant understand that it is constant and cannot be modified.

See also

- [The C# type system](#)

Properties (C# Programming Guide)

12/28/2021 • 4 minutes to read • [Edit Online](#)

A property is a member that provides a flexible mechanism to read, write, or compute the value of a private field. Properties can be used as if they are public data members, but they are actually special methods called *accessors*. This enables data to be accessed easily and still helps promote the safety and flexibility of methods.

Properties overview

- Properties enable a class to expose a public way of getting and setting values, while hiding implementation or verification code.
- A `get` property accessor is used to return the property value, and a `set` property accessor is used to assign a new value. In C# 9 and later, an `init` property accessor is used to assign a new value only during object construction. These accessors can have different access levels. For more information, see [Restricting Accessor Accessibility](#).
- The `value` keyword is used to define the value being assigned by the `set` or `init` accessor.
- Properties can be *read-write* (they have both a `get` and a `set` accessor), *read-only* (they have a `get` accessor but no `set` accessor), or *write-only* (they have a `set` accessor, but no `get` accessor). Write-only properties are rare and are most commonly used to restrict access to sensitive data.
- Simple properties that require no custom accessor code can be implemented either as expression body definitions or as [auto-implemented properties](#).

Properties with backing fields

One basic pattern for implementing a property involves using a private backing field for setting and retrieving the property value. The `get` accessor returns the value of the private field, and the `set` accessor may perform some data validation before assigning a value to the private field. Both accessors may also perform some conversion or computation on the data before it is stored or returned.

The following example illustrates this pattern. In this example, the `TimePeriod` class represents an interval of time. Internally, the class stores the time interval in seconds in a private field named `_seconds`. A read-write property named `Hours` allows the customer to specify the time interval in hours. Both the `get` and the `set` accessors perform the necessary conversion between hours and seconds. In addition, the `set` accessor validates the data and throws an [ArgumentOutOfRangeException](#) if the number of hours is invalid.


```

using System;

class TimePeriod
{
    private double _seconds;

    public double Hours
    {
        get { return _seconds / 3600; }
        set {
            if (value < 0 || value > 24)
                throw new ArgumentOutOfRangeException(
                    $"{nameof(value)} must be between 0 and 24.");

            _seconds = value * 3600;
        }
    }
}

class Program
{
    static void Main()
    {
        TimePeriod t = new TimePeriod();
        // The property assignment causes the 'set' accessor to be called.
        t.Hours = 24;

        // Retrieving the property causes the 'get' accessor to be called.
        Console.WriteLine($"Time in hours: {t.Hours}");
    }
}
// The example displays the following output:
//      Time in hours: 24

```

Expression body definitions

Property accessors often consist of single-line statements that just assign or return the result of an expression. You can implement these properties as expression-bodied members. Expression body definitions consist of the `=>` symbol followed by the expression to assign to or retrieve from the property.

Starting with C# 6, read-only properties can implement the `get` accessor as an expression-bodied member. In this case, neither the `get` accessor keyword nor the `return` keyword is used. The following example implements the read-only `Name` property as an expression-bodied member.

```
using System;

public class Person
{
    private string _firstName;
    private string _lastName;

    public Person(string first, string last)
    {
        _firstName = first;
        _lastName = last;
    }

    public string Name => $"{_firstName} {_lastName}";
}

public class Example
{
    public static void Main()
    {
        var person = new Person("Magnus", "Hedlund");
        Console.WriteLine(person.Name);
    }
}

// The example displays the following output:
//      Magnus Hedlund
```

Starting with C# 7.0, both the `get` and the `set` accessor can be implemented as expression-bodied members. In this case, the `get` and `set` keywords must be present. The following example illustrates the use of expression body definitions for both accessors. Note that the `return` keyword is not used with the `get` accessor.

```

using System;

public class SaleItem
{
    string _name;
    decimal _cost;

    public SaleItem(string name, decimal cost)
    {
        _name = name;
        _cost = cost;
    }

    public string Name
    {
        get => _name;
        set => _name = value;
    }

    public decimal Price
    {
        get => _cost;
        set => _cost = value;
    }
}

class Program
{
    static void Main(string[] args)
    {
        var item = new SaleItem("Shoes", 19.95m);
        Console.WriteLine($"{item.Name}: sells for {item.Price:C2}");
    }
}

// The example displays output like the following:
//      Shoes: sells for $19.95

```

Auto-implemented properties

In some cases, property `get` and `set` accessors just assign a value to or retrieve a value from a backing field without including any additional logic. By using auto-implemented properties, you can simplify your code while having the C# compiler transparently provide the backing field for you.

If a property has both a `get` and a `set` (or a `get` and an `init`) accessor, both must be auto-implemented. You define an auto-implemented property by using the `get` and `set` keywords without providing any implementation. The following example repeats the previous one, except that `Name` and `Price` are auto-implemented properties. The example also removes the parameterized constructor, so that `SaleItem` objects are now initialized with a call to the parameterless constructor and an [object initializer](#).

```
using System;

public class SaleItem
{
    public string Name
    { get; set; }

    public decimal Price
    { get; set; }
}

class Program
{
    static void Main(string[] args)
    {
        var item = new SaleItem{ Name = "Shoes", Price = 19.95m };
        Console.WriteLine($"{item.Name}: sells for {item.Price:C2}");
    }
}

// The example displays output like the following:
//      Shoes: sells for $19.95
```

Related sections

- [Using Properties](#)
- [Interface Properties](#)
- [Comparison Between Properties and Indexers](#)
- [Restricting Accessor Accessibility](#)
- [Auto-Implemented Properties](#)

C# Language Specification

For more information, see [Properties](#) in the [C# Language Specification](#). The language specification is the definitive source for C# syntax and usage.

See also

- [C# Programming Guide](#)
- [Using Properties](#)
- [Indexers](#)
- [get keyword](#)
- [set keyword](#)

Using Properties (C# Programming Guide)

12/28/2021 • 8 minutes to read • [Edit Online](#)

Properties combine aspects of both fields and methods. To the user of an object, a property appears to be a field, accessing the property requires the same syntax. To the implementer of a class, a property is one or two code blocks, representing a [get](#) accessor and/or a [set](#) accessor. The code block for the `get` accessor is executed when the property is read; the code block for the `set` accessor is executed when the property is assigned a new value. A property without a `set` accessor is considered read-only. A property without a `get` accessor is considered write-only. A property that has both accessors is read-write. In C# 9 and later, you can use an `init` accessor instead of a `set` accessor to make the property read-only.

Unlike fields, properties are not classified as variables. Therefore, you cannot pass a property as a [ref](#) or [out](#) parameter.

Properties have many uses: they can validate data before allowing a change; they can transparently expose data on a class where that data is actually retrieved from some other source, such as a database; they can take an action when data is changed, such as raising an event, or changing the value of other fields.

Properties are declared in the class block by specifying the access level of the field, followed by the type of the property, followed by the name of the property, and followed by a code block that declares a `get`-accessor and/or a `set` accessor. For example:

```
public class Date
{
    private int _month = 7; // Backing store

    public int Month
    {
        get => _month;
        set
        {
            if ((value > 0) && (value < 13))
            {
                _month = value;
            }
        }
    }
}
```

In this example, `Month` is declared as a property so that the `set` accessor can make sure that the `Month` value is set between 1 and 12. The `Month` property uses a private field to track the actual value. The real location of a property's data is often referred to as the property's "backing store." It is common for properties to use private fields as a backing store. The field is marked private in order to make sure that it can only be changed by calling the property. For more information about public and private access restrictions, see [Access Modifiers](#).

Auto-implemented properties provide simplified syntax for simple property declarations. For more information, see [Auto-Implemented Properties](#).

The get accessor

The body of the `get` accessor resembles that of a method. It must return a value of the property type. The execution of the `get` accessor is equivalent to reading the value of the field. For example, when you are returning the private variable from the `get` accessor and optimizations are enabled, the call to the `get`

accessor method is inlined by the compiler so there is no method-call overhead. However, a virtual `get` accessor method cannot be inlined because the compiler does not know at compile-time which method may actually be called at run time. The following is a `get` accessor that returns the value of a private field `_name`:

```
class Person
{
    private string _name; // the name field
    public string Name => _name; // the Name property
}
```

When you reference the property, except as the target of an assignment, the `get` accessor is invoked to read the value of the property. For example:

```
Person person = new Person();
//...

System.Console.Write(person.Name); // the get accessor is invoked here
```

The `get` accessor must end in a `return` or `throw` statement, and control cannot flow off the accessor body.

It is a bad programming style to change the state of the object by using the `get` accessor. For example, the following accessor produces the side effect of changing the state of the object every time that the `_number` field is accessed.

```
private int _number;
public int Number => _number++; // Don't do this
```

The `get` accessor can be used to return the field value or to compute it and return it. For example:

```
class Employee
{
    private string _name;
    public string Name => _name != null ? _name : "NA";
}
```

In the previous code segment, if you do not assign a value to the `Name` property, it will return the value `NA`.

The set accessor

The `set` accessor resembles a method whose return type is `void`. It uses an implicit parameter called `value`, whose type is the type of the property. In the following example, a `set` accessor is added to the `Name` property:

```
class Person
{
    private string _name; // the name field
    public string Name // the Name property
    {
        get => _name;
        set => _name = value;
    }
}
```

When you assign a value to the property, the `set` accessor is invoked by using an argument that provides the new value. For example:

```
Person person = new Person();
person.Name = "Joe"; // the set accessor is invoked here

System.Console.WriteLine(person.Name); // the get accessor is invoked here
```

It is an error to use the implicit parameter name, `value`, for a local variable declaration in a `set` accessor.

The init accessor

The code to create an `init` accessor is the same as the code to create a `set` accessor except that you use the `init` keyword instead of `set`. The difference is that the `init` accessor can only be used in the constructor or by using an [object-initializer](#).

Remarks

Properties can be marked as `public`, `private`, `protected`, `internal`, `protected internal`, or `private protected`. These access modifiers define how users of the class can access the property. The `get` and `set` accessors for the same property may have different access modifiers. For example, the `get` may be `public` to allow read-only access from outside the type, and the `set` may be `private` or `protected`. For more information, see [Access Modifiers](#).

A property may be declared as a static property by using the `static` keyword. This makes the property available to callers at any time, even if no instance of the class exists. For more information, see [Static Classes and Static Class Members](#).

A property may be marked as a virtual property by using the [virtual](#) keyword. This enables derived classes to override the property behavior by using the [override](#) keyword. For more information about these options, see [Inheritance](#).

A property overriding a virtual property can also be [sealed](#), specifying that for derived classes it is no longer virtual. Lastly, a property can be declared [abstract](#). This means that there is no implementation in the class, and derived classes must write their own implementation. For more information about these options, see [Abstract and Sealed Classes and Class Members](#).

NOTE

It is an error to use a [virtual](#), [abstract](#), or [override](#) modifier on an accessor of a `static` property.

Examples

This example demonstrates instance, static, and read-only properties. It accepts the name of the employee from the keyboard, increments `NumberOfEmployees` by 1, and displays the Employee name and number.

```

public class Employee
{
    public static int NumberOfEmployees;
    private static int _counter;
    private string _name;

    // A read-write instance property:
    public string Name
    {
        get => _name;
        set => _name = value;
    }

    // A read-only static property:
    public static int Counter => _counter;

    // A Constructor:
    public Employee() => _counter = ++NumberOfEmployees; // Calculate the employee's number:
}

class TestEmployee
{
    static void Main()
    {
        Employee.NumberOfEmployees = 107;
        Employee e1 = new Employee();
        e1.Name = "Claude Vige";

        System.Console.WriteLine("Employee number: {0}", Employee.Counter);
        System.Console.WriteLine("Employee name: {0}", e1.Name);
    }
}

/* Output:
    Employee number: 108
    Employee name: Claude Vige
*/

```

Hidden property example

This example demonstrates how to access a property in a base class that is hidden by another property that has the same name in a derived class:


```

public class Employee
{
    private string _name;
    public string Name
    {
        get => _name;
        set => _name = value;
    }
}

public class Manager : Employee
{
    private string _name;

    // Notice the use of the new modifier:
    public new string Name
    {
        get => _name;
        set => _name = value + ", Manager";
    }
}

class TestHiding
{
    static void Main()
    {
        Manager m1 = new Manager();

        // Derived class property.
        m1.Name = "John";

        // Base class property.
        ((Employee)m1).Name = "Mary";

        System.Console.WriteLine("Name in the derived class is: {0}", m1.Name);
        System.Console.WriteLine("Name in the base class is: {0}", ((Employee)m1).Name);
    }
}

/* Output:
    Name in the derived class is: John, Manager
    Name in the base class is: Mary
*/

```

The following are important points in the previous example:

- The property `Name` in the derived class hides the property `Name` in the base class. In such a case, the `new` modifier is used in the declaration of the property in the derived class:

```
public new string Name
```

- The cast `(Employee)` is used to access the hidden property in the base class:

```
((Employee)m1).Name = "Mary";
```

For more information about hiding members, see the [new Modifier](#).

Override property example

In this example, two classes, `Cube` and `Square`, implement an abstract class, `Shape`, and override its abstract `Area` property. Note the use of the [override](#) modifier on the properties. The program accepts the side as an input and calculates the areas for the square and cube. It also accepts the area as an input and calculates the

corresponding side for the square and cube.

```
abstract class Shape
{
    public abstract double Area
    {
        get;
        set;
    }
}

class Square : Shape
{
    public double side;

    //constructor
    public Square(double s) => side = s;

    public override double Area
    {
        get => side * side;
        set => side = System.Math.Sqrt(value);
    }
}

class Cube : Shape
{
    public double side;

    //constructor
    public Cube(double s) => side = s;

    public override double Area
    {
        get => 6 * side * side;
        set => side = System.Math.Sqrt(value / 6);
    }
}

class TestShapes
{
    static void Main()
    {
        // Input the side:
        System.Console.Write("Enter the side: ");
        double side = double.Parse(System.Console.ReadLine());

        // Compute the areas:
        Square s = new Square(side);
        Cube c = new Cube(side);

        // Display the results:
        System.Console.WriteLine("Area of the square = {0:F2}", s.Area);
        System.Console.WriteLine("Area of the cube = {0:F2}", c.Area);
        System.Console.WriteLine();

        // Input the area:
        System.Console.Write("Enter the area: ");
        double area = double.Parse(System.Console.ReadLine());

        // Compute the sides:
        s.Area = area;
        c.Area = area;

        // Display the results:
        System.Console.WriteLine("Side of the square = {0:F2}", s.side);
        System.Console.WriteLine("Side of the cube = {0:F2}", c.side);
    }
}
```

```
}  
/* Example Output:  
    Enter the side: 4  
    Area of the square = 16.00  
    Area of the cube = 96.00  
  
    Enter the area: 24  
    Side of the square = 4.90  
    Side of the cube = 2.00  
*/
```

See also

- [C# Programming Guide](#)
- [Properties](#)
- [Interface Properties](#)
- [Auto-Implemented Properties](#)

Interface Properties (C# Programming Guide)

12/28/2021 • 2 minutes to read • [Edit Online](#)

Properties can be declared on an [interface](#). The following example declares an interface property accessor:

```
public interface ISampleInterface
{
    // Property declaration:
    string Name
    {
        get;
        set;
    }
}
```

Interface properties typically don't have a body. The accessors indicate whether the property is read-write, read-only, or write-only. Unlike in classes and structs, declaring the accessors without a body doesn't declare an [auto-implemented property](#). Beginning with C# 8.0, an interface may define a default implementation for members, including properties. Defining a default implementation for a property in an interface is rare because interfaces may not define instance data fields.

Example

In this example, the interface `IEmployee` has a read-write property, `Name`, and a read-only property, `Counter`. The class `Employee` implements the `IEmployee` interface and uses these two properties. The program reads the name of a new employee and the current number of employees and displays the employee name and the computed employee number.

You could use the fully qualified name of the property, which references the interface in which the member is declared. For example:

```
string IEmployee.Name
{
    get { return "Employee Name"; }
    set { }
}
```

The preceding example demonstrates [Explicit Interface Implementation](#). For example, if the class `Employee` is implementing two interfaces `ICitizen` and `IEmployee` and both interfaces have the `Name` property, the explicit interface member implementation will be necessary. That is, the following property declaration:

```
string IEmployee.Name
{
    get { return "Employee Name"; }
    set { }
}
```

implements the `Name` property on the `IEmployee` interface, while the following declaration:

```

string ICitizen.Name
{
    get { return "Citizen Name"; }
    set { }
}

```

implements the `Name` property on the `ICitizen` interface.

```

interface IEmployee
{
    string Name
    {
        get;
        set;
    }

    int Counter
    {
        get;
    }
}

public class Employee : IEmployee
{
    public static int numberOfEmployees;

    private string _name;
    public string Name // read-write instance property
    {
        get => _name;
        set => _name = value;
    }

    private int _counter;
    public int Counter // read-only instance property
    {
        get => _counter;
    }

    // constructor
    public Employee() => _counter = ++numberOfEmployees;
}

```

```

System.Console.WriteLine("Enter number of employees: ");
Employee.numberOfEmployees = int.Parse(System.Console.ReadLine());

Employee e1 = new Employee();
System.Console.WriteLine("Enter the name of the new employee: ");
e1.Name = System.Console.ReadLine();

System.Console.WriteLine("The employee information:");
System.Console.WriteLine("Employee number: {0}", e1.Counter);
System.Console.WriteLine("Employee name: {0}", e1.Name);

```

Sample output

```
Enter number of employees: 210
Enter the name of the new employee: Hazem Abolrous
The employee information:
Employee number: 211
Employee name: Hazem Abolrous
```

See also

- [C# Programming Guide](#)
- [Properties](#)
- [Using Properties](#)
- [Comparison Between Properties and Indexers](#)
- [Indexers](#)
- [Interfaces](#)

Restricting Accessor Accessibility (C# Programming Guide)

12/28/2021 • 4 minutes to read • [Edit Online](#)

The `get` and `set` portions of a property or indexer are called *accessors*. By default these accessors have the same visibility or access level of the property or indexer to which they belong. For more information, see [accessibility levels](#). However, it is sometimes useful to restrict access to one of these accessors. Typically, this involves restricting the accessibility of the `set` accessor, while keeping the `get` accessor publicly accessible. For example:

```
private string _name = "Hello";

public string Name
{
    get
    {
        return _name;
    }
    protected set
    {
        _name = value;
    }
}
```

In this example, a property called `Name` defines a `get` and `set` accessor. The `get` accessor receives the accessibility level of the property itself, `public` in this case, while the `set` accessor is explicitly restricted by applying the `protected` access modifier to the accessor itself.

Restrictions on Access Modifiers on Accessors

Using the accessor modifiers on properties or indexers is subject to these conditions:

- You cannot use accessor modifiers on an interface or an explicit [interface](#) member implementation.
- You can use accessor modifiers only if the property or indexer has both `set` and `get` accessors. In this case, the modifier is permitted on only one of the two accessors.
- If the property or indexer has an [override](#) modifier, the accessor modifier must match the accessor of the overridden accessor, if any.
- The accessibility level on the accessor must be more restrictive than the accessibility level on the property or indexer itself.

Access Modifiers on Overriding Accessors

When you override a property or indexer, the overridden accessors must be accessible to the overriding code. Also, the accessibility of both the property/indexer and its accessors must match the corresponding overridden property/indexer and its accessors. For example:

```

public class Parent
{
    public virtual int TestProperty
    {
        // Notice the accessor accessibility level.
        protected set { }

        // No access modifier is used here.
        get { return 0; }
    }
}
public class Kid : Parent
{
    public override int TestProperty
    {
        // Use the same accessibility level as in the overridden accessor.
        protected set { }

        // Cannot use access modifier here.
        get { return 0; }
    }
}

```

Implementing Interfaces

When you use an accessor to implement an interface, the accessor may not have an access modifier. However, if you implement the interface using one accessor, such as `get`, the other accessor can have an access modifier, as in the following example:

```

public interface ISomeInterface
{
    int TestProperty
    {
        // No access modifier allowed here
        // because this is an interface.
        get;
    }
}

public class TestClass : ISomeInterface
{
    public int TestProperty
    {
        // Cannot use access modifier here because
        // this is an interface implementation.
        get { return 10; }

        // Interface property does not have set accessor,
        // so access modifier is allowed.
        protected set { }
    }
}

```

Accessor Accessibility Domain

If you use an access modifier on the accessor, the [accessibility domain](#) of the accessor is determined by this modifier.

If you did not use an access modifier on the accessor, the accessibility domain of the accessor is determined by the accessibility level of the property or indexer.

Example

The following example contains three classes, `BaseClass`, `DerivedClass`, and `MainClass`. There are two properties on the `BaseClass`, `Name` and `Id` on both classes. The example demonstrates how the property `Id` on `DerivedClass` can be hidden by the property `Id` on `BaseClass` when you use a restrictive access modifier such as `protected` or `private`. Therefore, when you assign values to this property, the property on the `BaseClass` class is called instead. Replacing the access modifier by `public` will make the property accessible.

The example also demonstrates that a restrictive access modifier, such as `private` or `protected`, on the `set` accessor of the `Name` property in `DerivedClass` prevents access to the accessor and generates an error when you assign to it.

```
public class BaseClass
{
    private string _name = "Name-BaseClass";
    private string _id = "ID-BaseClass";

    public string Name
    {
        get { return _name; }
        set { }
    }

    public string Id
    {
        get { return _id; }
        set { }
    }
}

public class DerivedClass : BaseClass
{
    private string _name = "Name-DerivedClass";
    private string _id = "ID-DerivedClass";

    new public string Name
    {
        get
        {
            return _name;
        }

        // Using "protected" would make the set accessor not accessible.
        set
        {
            _name = value;
        }
    }

    // Using private on the following property hides it in the Main Class.
    // Any assignment to the property will use Id in BaseClass.
    new private string Id
    {
        get
        {
            return _id;
        }
        set
        {
            _id = value;
        }
    }
}

class MainClass
```

```

{
    static void Main()
    {
        BaseClass b1 = new BaseClass();
        DerivedClass d1 = new DerivedClass();

        b1.Name = "Mary";
        d1.Name = "John";

        b1.Id = "Mary123";
        d1.Id = "John123"; // The BaseClass.Id property is called.

        System.Console.WriteLine("Base: {0}, {1}", b1.Name, b1.Id);
        System.Console.WriteLine("Derived: {0}, {1}", d1.Name, d1.Id);

        // Keep the console window open in debug mode.
        System.Console.WriteLine("Press any key to exit.");
        System.Console.ReadKey();
    }
}
/* Output:
    Base: Name-BaseClass, ID-BaseClass
    Derived: John, ID-BaseClass
*/

```

Comments

Notice that if you replace the declaration `new private string Id` by `new public string Id`, you get the output:

```
Name and ID in the base class: Name-BaseClass, ID-BaseClass
```

```
Name and ID in the derived class: John, John123
```

See also

- [C# Programming Guide](#)
- [Properties](#)
- [Indexers](#)
- [Access Modifiers](#)

How to declare and use read write properties (C# Programming Guide)

12/28/2021 • 2 minutes to read • [Edit Online](#)

Properties provide the convenience of public data members without the risks that come with unprotected, uncontrolled, and unverified access to an object's data. This is accomplished through *accessors*: special methods that assign and retrieve values from the underlying data member. The **set** accessor enables data members to be assigned, and the **get** accessor retrieves data member values.

This sample shows a `Person` class that has two properties: `Name` (string) and `Age` (int). Both properties provide `get` and `set` accessors, so they are considered read/write properties.

Example

```
class Person
{
    private string _name = "N/A";
    private int _age = 0;

    // Declare a Name property of type string:
    public string Name
    {
        get
        {
            return _name;
        }
        set
        {
            _name = value;
        }
    }

    // Declare an Age property of type int:
    public int Age
    {
        get
        {
            return _age;
        }

        set
        {
            _age = value;
        }
    }

    public override string ToString()
    {
        return "Name = " + Name + ", Age = " + Age;
    }
}

class TestPerson
{
    static void Main()
    {
        // Create a new Person object:
        Person person = new Person();
    }
}
```

```

        // Print out the name and the age associated with the person:
        Console.WriteLine("Person details - {0}", person);

        // Set some values on the person object:
        person.Name = "Joe";
        person.Age = 99;
        Console.WriteLine("Person details - {0}", person);

        // Increment the Age property:
        person.Age += 1;
        Console.WriteLine("Person details - {0}", person);

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
/* Output:
    Person details - Name = N/A, Age = 0
    Person details - Name = Joe, Age = 99
    Person details - Name = Joe, Age = 100
*/

```

Robust Programming

In the previous example, the `Name` and `Age` properties are [public](#) and include both a `get` and a `set` accessor. This allows any object to read and write these properties. It is sometimes desirable, however, to exclude one of the accessors. Omitting the `set` accessor, for example, makes the property read-only:

```

public string Name
{
    get
    {
        return _name;
    }
    set
    {
        _name = value;
    }
}

```

Alternatively, you can expose one accessor publicly but make the other private or protected. For more information, see [Asymmetric Accessor Accessibility](#).

Once the properties are declared, they can be used as if they were fields of the class. This allows for a very natural syntax when both getting and setting the value of a property, as in the following statements:

```

person.Name = "Joe";
person.Age = 99;

```

Note that in a property `set` method a special `value` variable is available. This variable contains the value that the user specified, for example:

```

_name = value;

```

Notice the clean syntax for incrementing the `Age` property on a `Person` object:

```
person.Age += 1;
```

If separate `set` and `get` methods were used to model properties, the equivalent code might look like this:

```
person.SetAge(person.GetAge() + 1);
```

The `ToString` method is overridden in this example:

```
public override string ToString()
{
    return "Name = " + Name + ", Age = " + Age;
}
```

Notice that `ToString` is not explicitly used in the program. It is invoked by default by the `WriteLine` calls.

See also

- [C# Programming Guide](#)
- [Properties](#)
- [The C# type system](#)

Auto-Implemented Properties (C# Programming Guide)

12/28/2021 • 2 minutes to read • [Edit Online](#)

In C# 3.0 and later, auto-implemented properties make property-declaration more concise when no additional logic is required in the property accessors. They also enable client code to create objects. When you declare a property as shown in the following example, the compiler creates a private, anonymous backing field that can only be accessed through the property's `get` and `set` accessors. In C# 9 and later, `init` accessors can also be declared as auto-implemented properties.

Example

The following example shows a simple class that has some auto-implemented properties:

```
// This class is mutable. Its data can be modified from
// outside the class.
class Customer
{
    // Auto-implemented properties for trivial get and set
    public double TotalPurchases { get; set; }
    public string Name { get; set; }
    public int CustomerId { get; set; }

    // Constructor
    public Customer(double purchases, string name, int id)
    {
        TotalPurchases = purchases;
        Name = name;
        CustomerId = id;
    }

    // Methods
    public string GetContactInfo() { return "ContactInfo"; }
    public string GetTransactionHistory() { return "History"; }

    // .. Additional methods, events, etc.
}

class Program
{
    static void Main()
    {
        // Initialize a new object.
        Customer cust1 = new Customer(4987.63, "Northwind", 90108);

        // Modify a property.
        cust1.TotalPurchases += 499.99;
    }
}
```

You can't declare auto-implemented properties in interfaces. Auto-implemented properties declare a private instance backing field, and interfaces may not declare instance fields. Declaring a property in an interface without defining a body declares a property with accessors that must be implemented by each type that implements that interface.

In C# 6 and later, you can initialize auto-implemented properties similarly to fields:

```
public string FirstName { get; set; } = "Jane";
```

The class that is shown in the previous example is mutable. Client code can change the values in objects after creation. In complex classes that contain significant behavior (methods) as well as data, it's often necessary to have public properties. However, for small classes or structs that just encapsulate a set of values (data) and have little or no behaviors, you should use one of the following options for making the objects immutable:

- Declare only a `get` accessor (immutable everywhere except the constructor).
- Declare a `get` accessor and an `init` accessor (immutable everywhere except during object construction).
- Declare the `set` accessor as `private` (immutable to consumers).

For more information, see [How to implement a lightweight class with auto-implemented properties](#).

See also

- [Properties](#)
- [Modifiers](#)

How to implement a lightweight class with auto-implemented properties (C# Programming Guide)

12/28/2021 • 3 minutes to read • [Edit Online](#)

This example shows how to create an immutable lightweight class that serves only to encapsulate a set of auto-implemented properties. Use this kind of construct instead of a struct when you must use reference type semantics.

You can make an immutable property in the following ways:

- Declare only the `get` accessor, which makes the property immutable everywhere except in the type's constructor.
- Declare an `init` accessor instead of a `set` accessor, which makes the property settable only in the constructor or by using an `object initializer`.
- Declare the `set` accessor to be `private`. The property is settable within the type, but it is immutable to consumers.

When you declare a private `set` accessor, you cannot use an object initializer to initialize the property. You must use a constructor or a factory method.

The following example shows how a property with only get accessor differs than one with get and private set.

```
class Contact
{
    public string Name { get; }
    public string Address { get; private set; }

    public Contact(string contactName, string contactAddress)
    {
        // Both properties are accessible in the constructor.
        Name = contactName;
        Address = contactAddress;
    }

    // Name isn't assignable here. This will generate a compile error.
    //public void ChangeName(string newName) => Name = newName;

    // Address is assignable here.
    public void ChangeAddress(string newAddress) => Address = newAddress;
}
```

Example

The following example shows two ways to implement an immutable class that has auto-implemented properties. Each way declares one of the properties with a private `set` and one of the properties with a `get` only. The first class uses a constructor only to initialize the properties, and the second class uses a static factory method that calls a constructor.

```
// This class is immutable. After an object is created,
// it cannot be modified from outside the class. It uses a
// constructor to initialize its properties.
class Contact
{
    // ...
}
```



```

    // Read-only property.
    public string Name { get; }

    // Read-write property with a private set accessor.
    public string Address { get; private set; }

    // Public constructor.
    public Contact(string contactName, string contactAddress)
    {
        Name = contactName;
        Address = contactAddress;
    }
}

// This class is immutable. After an object is created,
// it cannot be modified from outside the class. It uses a
// static method and private constructor to initialize its properties.
public class Contact2
{
    // Read-write property with a private set accessor.
    public string Name { get; private set; }

    // Read-only property.
    public string Address { get; }

    // Private constructor.
    private Contact2(string contactName, string contactAddress)
    {
        Name = contactName;
        Address = contactAddress;
    }

    // Public factory method.
    public static Contact2 CreateContact(string name, string address)
    {
        return new Contact2(name, address);
    }
}

public class Program
{
    static void Main()
    {
        // Some simple data sources.
        string[] names = {"Terry Adams", "Fadi Fakhouri", "Hanying Feng",
            "Cesar Garcia", "Debra Garcia"};
        string[] addresses = {"123 Main St.", "345 Cypress Ave.", "678 1st Ave",
            "12 108th St.", "89 E. 42nd St."};

        // Simple query to demonstrate object creation in select clause.
        // Create Contact objects by using a constructor.
        var query1 = from i in Enumerable.Range(0, 5)
            select new Contact(names[i], addresses[i]);

        // List elements cannot be modified by client code.
        var list = query1.ToList();
        foreach (var contact in list)
        {
            Console.WriteLine("{0}, {1}", contact.Name, contact.Address);
        }

        // Create Contact2 objects by using a static factory method.
        var query2 = from i in Enumerable.Range(0, 5)
            select Contact2.CreateContact(names[i], addresses[i]);

        // Console output is identical to query1.
        var list2 = query2.ToList();
    }
}

```

```

        // List elements cannot be modified by client code.
        // CS0272:
        // list2[0].Name = "Eugene Zabokritski";

        // Keep the console open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}

/* Output:
    Terry Adams, 123 Main St.
    Fadi Fakhouri, 345 Cypress Ave.
    Hanying Feng, 678 1st Ave
    Cesar Garcia, 12 108th St.
    Debra Garcia, 89 E. 42nd St.
*/

```

The compiler creates backing fields for each auto-implemented property. The fields are not accessible directly from source code.

See also

- [Properties](#)
- [struct](#)
- [Object and Collection Initializers](#)

Methods (C# Programming Guide)

12/28/2021 • 10 minutes to read • [Edit Online](#)

A method is a code block that contains a series of statements. A program causes the statements to be executed by calling the method and specifying any required method arguments. In C#, every executed instruction is performed in the context of a method.

The `Main` method is the entry point for every C# application and it's called by the common language runtime (CLR) when the program is started. In an application that uses [top-level statements](#), the `Main` method is generated by the compiler and contains all top-level statements.

NOTE

This article discusses named methods. For information about anonymous functions, see [Lambda expressions](#).

Method signatures

Methods are declared in a [class](#), [struct](#), or [interface](#) by specifying the access level such as `public` or `private`, optional modifiers such as `abstract` or `sealed`, the return value, the name of the method, and any method parameters. These parts together are the signature of the method.

IMPORTANT

A return type of a method is not part of the signature of the method for the purposes of method overloading. However, it is part of the signature of the method when determining the compatibility between a delegate and the method that it points to.

Method parameters are enclosed in parentheses and are separated by commas. Empty parentheses indicate that the method requires no parameters. This class contains four methods:

```
abstract class Motorcycle
{
    // Anyone can call this.
    public void StartEngine() { /* Method statements here */ }

    // Only derived classes can call this.
    protected void AddGas(int gallons) { /* Method statements here */ }

    // Derived classes can override the base class implementation.
    public virtual int Drive(int miles, int speed) { /* Method statements here */ return 1; }

    // Derived classes must implement this.
    public abstract double GetTopSpeed();
}
```

Method access

Calling a method on an object is like accessing a field. After the object name, add a period, the name of the method, and parentheses. Arguments are listed within the parentheses, and are separated by commas. The methods of the `Motorcycle` class can therefore be called as in the following example:

```

class TestMotorcycle : Motorcycle
{
    public override double GetTopSpeed()
    {
        return 108.4;
    }

    static void Main()
    {
        TestMotorcycle moto = new TestMotorcycle();

        moto.StartEngine();
        moto.AddGas(15);
        moto.Drive(5, 20);
        double speed = moto.GetTopSpeed();
        Console.WriteLine("My top speed is {0}", speed);
    }
}

```

Method parameters vs. arguments

The method definition specifies the names and types of any parameters that are required. When calling code calls the method, it provides concrete values called arguments for each parameter. The arguments must be compatible with the parameter type but the argument name (if any) used in the calling code doesn't have to be the same as the parameter named defined in the method. For example:

```

public void Caller()
{
    int numA = 4;
    // Call with an int variable.
    int productA = Square(numA);

    int numB = 32;
    // Call with another int variable.
    int productB = Square(numB);

    // Call with an integer literal.
    int productC = Square(12);

    // Call with an expression that evaluates to int.
    productC = Square(productA * 3);
}

int Square(int i)
{
    // Store input argument in a local variable.
    int input = i;
    return input * input;
}

```

Passing by reference vs. passing by value

By default, when an instance of a [value type](#) is passed to a method, its copy is passed instead of the instance itself. Therefore, changes to the argument have no effect on the original instance in the calling method. To pass a value-type instance by reference, use the `ref` keyword. For more information, see [Passing Value-Type Parameters](#).

When an object of a reference type is passed to a method, a reference to the object is passed. That is, the

method receives not the object itself but an argument that indicates the location of the object. If you change a member of the object by using this reference, the change is reflected in the argument in the calling method, even if you pass the object by value.

You create a reference type by using the `class` keyword, as the following example shows:

```
public class SampleRefType
{
    public int value;
}
```

Now, if you pass an object that is based on this type to a method, a reference to the object is passed. The following example passes an object of type `SampleRefType` to method `ModifyObject`:

```
public static void TestRefType()
{
    SampleRefType rt = new SampleRefType();
    rt.value = 44;
    ModifyObject(rt);
    Console.WriteLine(rt.value);
}

static void ModifyObject(SampleRefType obj)
{
    obj.value = 33;
}
```

The example does essentially the same thing as the previous example in that it passes an argument by value to a method. But, because a reference type is used, the result is different. The modification that is made in `ModifyObject` to the `value` field of the parameter, `obj`, also changes the `value` field of the argument, `rt`, in the `TestRefType` method. The `TestRefType` method displays 33 as the output.

For more information about how to pass reference types by reference and by value, see [Passing Reference-Type Parameters](#) and [Reference Types](#).

Return values

Methods can return a value to the caller. If the return type (the type listed before the method name) is not `void`, the method can return the value by using the [return statement](#). A statement with the `return` keyword followed by a value that matches the return type will return that value to the method caller.

The value can be returned to the caller by value or, starting with C# 7.0, [by reference](#). Values are returned to the caller by reference if the `ref` keyword is used in the method signature and it follows each `return` keyword. For example, the following method signature and return statement indicate that the method returns a variable named `estDistance` by reference to the caller.

```
public ref double GetEstimatedDistance()
{
    return ref estDistance;
}
```

The `return` keyword also stops the execution of the method. If the return type is `void`, a `return` statement without a value is still useful to stop the execution of the method. Without the `return` keyword, the method will stop executing when it reaches the end of the code block. Methods with a non-void return type are required to use the `return` keyword to return a value. For example, these two methods use the `return` keyword to return integers:

```

class SimpleMath
{
    public int AddTwoNumbers(int number1, int number2)
    {
        return number1 + number2;
    }

    public int SquareANumber(int number)
    {
        return number * number;
    }
}

```

To use a value returned from a method, the calling method can use the method call itself anywhere a value of the same type would be sufficient. You can also assign the return value to a variable. For example, the following two code examples accomplish the same goal:

```

int result = obj.AddTwoNumbers(1, 2);
result = obj.SquareANumber(result);
// The result is 9.
Console.WriteLine(result);

```

```

result = obj.SquareANumber(obj.AddTwoNumbers(1, 2));
// The result is 9.
Console.WriteLine(result);

```

Using a local variable, in this case, `result`, to store a value is optional. It may help the readability of the code, or it may be necessary if you need to store the original value of the argument for the entire scope of the method.

To use a value returned by reference from a method, you must declare a [ref local](#) variable if you intend to modify its value. For example, if the `Planet.GetEstimatedDistance` method returns a [Double](#) value by reference, you can define it as a ref local variable with code like the following:

```

ref int distance = Planet.GetEstimatedDistance();

```

Returning a multi-dimensional array from a method, `M`, that modifies the array's contents is not necessary if the calling function passed the array into `M`. You may return the resulting array from `M` for good style or functional flow of values, but it is not necessary because C# passes all reference types by value, and the value of an array reference is the pointer to the array. In the method `M`, any changes to the array's contents are observable by any code that has a reference to the array, as shown in the following example:

```

static void Main(string[] args)
{
    int[,] matrix = new int[2, 2];
    FillMatrix(matrix);
    // matrix is now full of -1
}

public static void FillMatrix(int[,] matrix)
{
    for (int i = 0; i < matrix.GetLength(0); i++)
    {
        for (int j = 0; j < matrix.GetLength(1); j++)
        {
            matrix[i, j] = -1;
        }
    }
}

```

Async methods

By using the async feature, you can invoke asynchronous methods without using explicit callbacks or manually splitting your code across multiple methods or lambda expressions.

If you mark a method with the `async` modifier, you can use the `await` operator in the method. When control reaches an await expression in the async method, control returns to the caller, and progress in the method is suspended until the awaited task completes. When the task is complete, execution can resume in the method.

NOTE

An async method returns to the caller when either it encounters the first awaited object that's not yet complete or it gets to the end of the async method, whichever occurs first.

An async method typically has a return type of `Task<TResult>`, `Task`, `IEnumerable<T>` or `void`. The `void` return type is used primarily to define event handlers, where a `void` return type is required. An async method that returns `void` can't be awaited, and the caller of a void-returning method can't catch exceptions that the method throws. Starting with C# 7.0, an async method can have [any task-like return type](#).

In the following example, `DelayAsync` is an async method that has a return type of `Task<TResult>`. `DelayAsync` has a `return` statement that returns an integer. Therefore the method declaration of `DelayAsync` must have a return type of `Task<int>`. Because the return type is `Task<int>`, the evaluation of the `await` expression in `DoSomethingAsync` produces an integer as the following statement demonstrates: `int result = await delayTask`.

The `Main` method is an example of an async method that has a return type of `Task`. It goes to the `DoSomethingAsync` method, and because it is expressed with a single line, it can omit the `async` and `await` keywords. Because `DoSomethingAsync` is an async method, the task for the call to `DoSomethingAsync` must be awaited, as the following statement shows: `await DoSomethingAsync();`.

```

using System;
using System.Threading.Tasks;

class Program
{
    static Task Main() => DoSomethingAsync();

    static async Task DoSomethingAsync()
    {
        Task<int> delayTask = DelayAsync();
        int result = await delayTask;

        // The previous two statements may be combined into
        // the following statement.
        //int result = await DelayAsync();

        Console.WriteLine($"Result: {result}");
    }

    static async Task<int> DelayAsync()
    {
        await Task.Delay(100);
        return 5;
    }
}
// Example output:
// Result: 5

```

An async method can't declare any [ref](#) or [out](#) parameters, but it can call methods that have such parameters.

For more information about async methods, see [Asynchronous programming with async and await](#) and [Async return types](#).

Expression body definitions

It is common to have method definitions that simply return immediately with the result of an expression, or that have a single statement as the body of the method. There is a syntax shortcut for defining such methods using

```
=> :
```

```

public Point Move(int dx, int dy) => new Point(x + dx, y + dy);
public void Print() => Console.WriteLine(First + " " + Last);
// Works with operators, properties, and indexers too.
public static Complex operator +(Complex a, Complex b) => a.Add(b);
public string Name => First + " " + Last;
public Customer this[long id] => store.LookupCustomer(id);

```

If the method returns `void` or is an async method, then the body of the method must be a statement expression (same as with lambdas). For properties and indexers, they must be read only, and you don't use the `get` accessor keyword.

Iterators

An iterator performs a custom iteration over a collection, such as a list or an array. An iterator uses the [yield return](#) statement to return each element one at a time. When a [yield return](#) statement is reached, the current location in code is remembered. Execution is restarted from that location when the iterator is called the next time.

You call an iterator from client code by using a [foreach](#) statement.

The return type of an iterator can be [IEnumerable](#), [IEnumerable<T>](#), [IEnumerator](#), or [IEnumerator<T>](#).

For more information, see [Iterators](#).

C# language specification

For more information, see the [C# Language Specification](#). The language specification is the definitive source for C# syntax and usage.

See also

- [C# Programming Guide](#)
- [The C# type system](#)
- [Access Modifiers](#)
- [Static Classes and Static Class Members](#)
- [Inheritance](#)
- [Abstract and Sealed Classes and Class Members](#)
- [params](#)
- [out](#)
- [ref](#)
- [Passing Parameters](#)

Local functions (C# Programming Guide)

12/28/2021 • 10 minutes to read • [Edit Online](#)

Starting with C# 7.0, C# supports *local functions*. Local functions are private methods of a type that are nested in another member. They can only be called from their containing member. Local functions can be declared in and called from:

- Methods, especially iterator methods and async methods
- Constructors
- Property accessors
- Event accessors
- Anonymous methods
- Lambda expressions
- Finalizers
- Other local functions

However, local functions can't be declared inside an expression-bodied member.

NOTE

In some cases, you can use a lambda expression to implement functionality also supported by a local function. For a comparison, see [Local functions vs. lambda expressions](#).

Local functions make the intent of your code clear. Anyone reading your code can see that the method is not callable except by the containing method. For team projects, they also make it impossible for another developer to mistakenly call the method directly from elsewhere in the class or struct.

Local function syntax

A local function is defined as a nested method inside a containing member. Its definition has the following syntax:

```
<modifiers> <return-type> <method-name> <parameter-list>
```

You can use the following modifiers with a local function:

- `async`
- `unsafe`
- `static` (in C# 8.0 and later). A static local function can't capture local variables or instance state.
- `extern` (in C# 9.0 and later). An external local function must be `static`.

All local variables that are defined in the containing member, including its method parameters, are accessible in a non-static local function.

Unlike a method definition, a local function definition cannot include the member access modifier. Because all local functions are private, including an access modifier, such as the `private` keyword, generates compiler error CS0106, "The modifier 'private' is not valid for this item."

The following example defines a local function named `AppendPathSeparator` that is private to a method named

GetText :

```
private static string GetText(string path, string filename)
{
    var reader = File.OpenText($"{AppendPathSeparator(path)}{filename}");
    var text = reader.ReadToEnd();
    return text;

    string AppendPathSeparator(string filepath)
    {
        return filepath.EndsWith(@"\") ? filepath : filepath + @"\";
    }
}
```

Beginning with C# 9.0, you can apply attributes to a local function, its parameters and type parameters, as the following example shows:

```
#nullable enable
private static void Process(string?[] lines, string mark)
{
    foreach (var line in lines)
    {
        if (IsValid(line))
        {
            // Processing logic...
        }
    }

    bool IsValid([NotNullWhen(true)] string? line)
    {
        return !string.IsNullOrEmpty(line) && line.Length >= mark.Length;
    }
}
```

The preceding example uses a [special attribute](#) to assist the compiler in static analysis in a nullable context.

Local functions and exceptions

One of the useful features of local functions is that they can allow exceptions to surface immediately. For method iterators, exceptions are surfaced only when the returned sequence is enumerated, and not when the iterator is retrieved. For async methods, any exceptions thrown in an async method are observed when the returned task is awaited.

The following example defines an `OddSequence` method that enumerates odd numbers in a specified range. Because it passes a number greater than 100 to the `OddSequence` enumerator method, the method throws an [ArgumentOutOfRangeException](#). As the output from the example shows, the exception surfaces only when you iterate the numbers, and not when you retrieve the enumerator.

```

using System;
using System.Collections.Generic;

public class IteratorWithoutLocalExample
{
    public static void Main()
    {
        IEnumerable<int> xs = OddSequence(50, 110);
        Console.WriteLine("Retrieved enumerator...");

        foreach (var x in xs) // line 11
        {
            Console.Write($"{x} ");
        }
    }

    public static IEnumerable<int> OddSequence(int start, int end)
    {
        if (start < 0 || start > 99)
            throw new ArgumentOutOfRangeException(nameof(start), "start must be between 0 and 99.");
        if (end > 100)
            throw new ArgumentOutOfRangeException(nameof(end), "end must be less than or equal to 100.");
        if (start >= end)
            throw new ArgumentException("start must be less than end.");

        for (int i = start; i <= end; i++)
        {
            if (i % 2 == 1)
                yield return i;
        }
    }
}

// The example displays the output like this:
//
// Retrieved enumerator...
// Unhandled exception. System.ArgumentOutOfRangeException: end must be less than or equal to 100.
// (Parameter 'end')
// at IteratorWithoutLocalExample.OddSequence(Int32 start, Int32 end)+MoveNext() in
// IteratorWithoutLocal.cs:line 22
// at IteratorWithoutLocalExample.Main() in IteratorWithoutLocal.cs:line 11

```

If you put iterator logic into a local function, argument validation exceptions are thrown when you retrieve the enumerator, as the following example shows:

```

using System;
using System.Collections.Generic;

public class IteratorWithLocalExample
{
    public static void Main()
    {
        IEnumerable<int> xs = OddSequence(50, 110); // line 8
        Console.WriteLine("Retrieved enumerator...");

        foreach (var x in xs)
        {
            Console.Write($"{x} ");
        }
    }

    public static IEnumerable<int> OddSequence(int start, int end)
    {
        if (start < 0 || start > 99)
            throw new ArgumentOutOfRangeException(nameof(start), "start must be between 0 and 99.");
        if (end > 100)
            throw new ArgumentOutOfRangeException(nameof(end), "end must be less than or equal to 100.");
        if (start >= end)
            throw new ArgumentException("start must be less than end.");

        return GetOddSequenceEnumerator();

        IEnumerable<int> GetOddSequenceEnumerator()
        {
            for (int i = start; i <= end; i++)
            {
                if (i % 2 == 1)
                    yield return i;
            }
        }
    }
}

// The example displays the output like this:
//
// Unhandled exception. System.ArgumentOutOfRangeException: end must be less than or equal to 100.
// (Parameter 'end')
// at IteratorWithLocalExample.OddSequence(Int32 start, Int32 end) in IteratorWithLocal.cs:line 22
// at IteratorWithLocalExample.Main() in IteratorWithLocal.cs:line 8

```

Local functions vs. lambda expressions

At first glance, local functions and [lambda expressions](#) are very similar. In many cases, the choice between using lambda expressions and local functions is a matter of style and personal preference. However, there are real differences in where you can use one or the other that you should be aware of.

Let's examine the differences between the local function and lambda expression implementations of the factorial algorithm. Here's the version using a local function:

```

public static int LocalFunctionFactorial(int n)
{
    return nthFactorial(n);

    int nthFactorial(int number) => number < 2
        ? 1
        : number * nthFactorial(number - 1);
}

```

This version uses lambda expressions:

```

public static int LambdaFactorial(int n)
{
    Func<int, int> nthFactorial = default(Func<int, int>);

    nthFactorial = number => number < 2
        ? 1
        : number * nthFactorial(number - 1);

    return nthFactorial(n);
}

```

Naming

Local functions are explicitly named like methods. Lambda expressions are anonymous methods and need to be assigned to variables of a `delegate` type, typically either `Action` or `Func` types. When you declare a local function, the process is like writing a normal method; you declare a return type and a function signature.

Function signatures and lambda expression types

Lambda expressions rely on the type of the `Action` / `Func` variable that they're assigned to determine the argument and return types. In local functions, since the syntax is much like writing a normal method, argument types and return type are already part of the function declaration.

Beginning with C# 10, some lambda expressions have a *natural type*, which enables the compiler to infer the return type and parameter types of the lambda expression.

Definite assignment

Lambda expressions are objects that are declared and assigned at run time. In order for a lambda expression to be used, it needs to be definitely assigned: the `Action` / `Func` variable that it will be assigned to must be declared and the lambda expression assigned to it. Notice that `LambdaFactorial` must declare and initialize the lambda expression `nthFactorial` before defining it. Not doing so results in a compile time error for referencing `nthFactorial` before assigning it.

Local functions are defined at compile time. As they're not assigned to variables, they can be referenced from any code location **where it is in scope**; in our first example `LocalFunctionFactorial`, we could declare our local function either above or below the `return` statement and not trigger any compiler errors.

These differences mean that recursive algorithms are easier to create using local functions. You can declare and define a local function that calls itself. Lambda expressions must be declared, and assigned a default value before they can be re-assigned to a body that references the same lambda expression.

Implementation as a delegate

Lambda expressions are converted to delegates when they're declared. Local functions are more flexible in that they can be written like a traditional method *or* as a delegate. Local functions are only converted to delegates when *used* as a delegate.

If you declare a local function and only reference it by calling it like a method, it will not be converted to a delegate.

Variable capture

The rules of [definite assignment](#) also affect any variables that are captured by the local function or lambda expression. The compiler can perform static analysis that enables local functions to definitely assign captured variables in the enclosing scope. Consider this example:

```
int M()
{
    int y;
    LocalFunction();
    return y;

    void LocalFunction() => y = 0;
}
```

The compiler can determine that `LocalFunction` definitely assigns `y` when called. Because `LocalFunction` is called before the `return` statement, `y` is definitely assigned at the `return` statement.

Note that when a local function captures variables in the enclosing scope, the local function is implemented as a delegate type.

Heap allocations

Depending on their use, local functions can avoid heap allocations that are always necessary for lambda expressions. If a local function is never converted to a delegate, and none of the variables captured by the local function are captured by other lambdas or local functions that are converted to delegates, the compiler can avoid heap allocations.

Consider this async example:

```
public async Task<string> PerformLongRunningWorkLambda(string address, int index, string name)
{
    if (string.IsNullOrEmpty(address))
        throw new ArgumentException(message: "An address is required", paramName: nameof(address));
    if (index < 0)
        throw new ArgumentOutOfRangeException(paramName: nameof(index), message: "The index must be non-negative");
    if (string.IsNullOrEmpty(name))
        throw new ArgumentException(message: "You must supply a name", paramName: nameof(name));

    Func<Task<string>> longRunningWorkImplementation = async () =>
    {
        var interimResult = await FirstWork(address);
        var secondResult = await SecondStep(index, name);
        return $"The results are {interimResult} and {secondResult}. Enjoy.";
    };

    return await longRunningWorkImplementation();
}
```

The closure for this lambda expression contains the `address`, `index` and `name` variables. In the case of local functions, the object that implements the closure may be a `struct` type. That struct type would be passed by reference to the local function. This difference in implementation would save on an allocation.

The instantiation necessary for lambda expressions means extra memory allocations, which may be a performance factor in time-critical code paths. Local functions do not incur this overhead. In the example above, the local functions version has two fewer allocations than the lambda expression version.

If you know that your local function won't be converted to a delegate and none of the variables captured by it are captured by other lambdas or local functions that are converted to delegates, you can guarantee that your local function avoids being allocated on the heap by declaring it as a `static` local function. Note that this feature is available in C# 8.0 and newer.

NOTE

The local function equivalent of this method also uses a class for the closure. Whether the closure for a local function is implemented as a `class` or a `struct` is an implementation detail. A local function may use a `struct` whereas a lambda will always use a `class`.

```
public async Task<string> PerformLongRunningWork(string address, int index, string name)
{
    if (string.IsNullOrEmpty(address))
        throw new ArgumentException(message: "An address is required", paramName: nameof(address));
    if (index < 0)
        throw new ArgumentOutOfRangeException(paramName: nameof(index), message: "The index must be non-negative");
    if (string.IsNullOrEmpty(name))
        throw new ArgumentException(message: "You must supply a name", paramName: nameof(name));

    return await longRunningWorkImplementation();

    async Task<string> longRunningWorkImplementation()
    {
        var interimResult = await FirstWork(address);
        var secondResult = await SecondStep(index, name);
        return $"The results are {interimResult} and {secondResult}. Enjoy.";
    }
}
```

Usage of the `yield` keyword

One final advantage not demonstrated in this sample is that local functions can be implemented as iterators, using the `yield return` syntax to produce a sequence of values.

```
public IEnumerable<string> SequenceToLowercase(IEnumerable<string> input)
{
    if (!input.Any())
    {
        throw new ArgumentException("There are no items to convert to lowercase.");
    }

    return LowercaseIterator();

    IEnumerable<string> LowercaseIterator()
    {
        foreach (var output in input.Select(item => item.ToLower()))
        {
            yield return output;
        }
    }
}
```

The `yield return` statement is not allowed in lambda expressions, see [compiler error CS1621](#).

While local functions may seem redundant to lambda expressions, they actually serve different purposes and have different uses. Local functions are more efficient for the case when you want to write a function that is called only from the context of another method.

See also

- [Methods](#)

Ref returns and ref locals

12/28/2021 • 7 minutes to read • [Edit Online](#)

Starting with C# 7.0, C# supports reference return values (ref returns). A reference return value allows a method to return a reference to a variable, rather than a value, back to a caller. The caller can then choose to treat the returned variable as if it were returned by value or by reference. The caller can create a new variable that is itself a reference to the returned value, called a ref local.

What is a reference return value?

Most developers are familiar with passing an argument to a called method *by reference*. A called method's argument list includes a variable passed by reference. Any changes made to its value by the called method are observed by the caller. A *reference return value* means that a method returns a *reference* (or an alias) to some variable. That variable's scope must include the method. That variable's lifetime must extend beyond the return of the method. Modifications to the method's return value by the caller are made to the variable that is returned by the method.

Declaring that a method returns a *reference return value* indicates that the method returns an alias to a variable. The design intent is often that the calling code should have access to that variable through the alias, including to modify it. It follows that methods returning by reference can't have the return type `void`.

There are some restrictions on the expression that a method can return as a reference return value. Restrictions include:

- The return value must have a lifetime that extends beyond the execution of the method. In other words, it cannot be a local variable in the method that returns it. It can be an instance or static field of a class, or it can be an argument passed to the method. Attempting to return a local variable generates compiler error CS8168, "Cannot return local 'obj' by reference because it is not a ref local."
- The return value cannot be the literal `null`. Returning `null` generates compiler error CS8156, "An expression cannot be used in this context because it may not be returned by reference."

A method with a ref return can return an alias to a variable whose value is currently the null (uninstantiated) value or a [nullable value type](#) for a value type.

- The return value cannot be a constant, an enumeration member, the by-value return value from a property, or a method of a `class` or `struct`. Violating this rule generates compiler error CS8156, "An expression cannot be used in this context because it may not be returned by reference."

In addition, reference return values are not allowed on async methods. An asynchronous method may return before it has finished execution, while its return value is still unknown.

Defining a ref return value

A method that returns a *reference return value* must satisfy the following two conditions:

- The method signature includes the [ref](#) keyword in front of the return type.
- Each [return](#) statement in the method body includes the [ref](#) keyword in front of the name of the returned instance.

The following example shows a method that satisfies those conditions and returns a reference to a `Person` object named `p`:

```
public ref Person GetContactInformation(string fname, string lname)
{
    // ...method implementation...
    return ref p;
}
```

Consuming a ref return value

The ref return value is an alias to another variable in the called method's scope. You can interpret any use of the ref return as using the variable it aliases:

- When you assign its value, you are assigning a value to the variable it aliases.
- When you read its value, you are reading the value of the variable it aliases.
- If you return it *by reference*, you are returning an alias to that same variable.
- If you pass it to another method *by reference*, you are passing a reference to the variable it aliases.
- When you make a [ref local](#) alias, you make a new alias to the same variable.

Ref locals

Assume the `GetContactInformation` method is declared as a ref return:

```
public ref Person GetContactInformation(string fname, string lname)
```

A by-value assignment reads the value of a variable and assigns it to a new variable:

```
Person p = contacts.GetContactInformation("Brandie", "Best");
```

The preceding assignment declares `p` as a local variable. Its initial value is copied from reading the value returned by `GetContactInformation`. Any future assignments to `p` will not change the value of the variable returned by `GetContactInformation`. The variable `p` is no longer an alias to the variable returned.

You declare a *ref local* variable to copy the alias to the original value. In the following assignment, `p` is an alias to the variable returned from `GetContactInformation`.

```
ref Person p = ref contacts.GetContactInformation("Brandie", "Best");
```

Subsequent usage of `p` is the same as using the variable returned by `GetContactInformation` because `p` is an alias for that variable. Changes to `p` also change the variable returned from `GetContactInformation`.

The `ref` keyword is used both before the local variable declaration *and* before the method call.

You can access a value by reference in the same way. In some cases, accessing a value by reference increases performance by avoiding a potentially expensive copy operation. For example, the following statement shows how one can define a ref local value that is used to reference a value.

```
ref VeryLargeStruct reflocal = ref veryLargeStruct;
```

The `ref` keyword is used both before the local variable declaration *and* before the value in the second example. Failure to include both `ref` keywords in the variable declaration and assignment in both examples results in compiler error CS8172, "Cannot initialize a by-reference variable with a value."

Prior to C# 7.3, ref local variables couldn't be reassigned to refer to different storage after being initialized. That

restriction has been removed. The following example shows a reassignment:

```
ref VeryLargeStruct refLocal = ref veryLargeStruct; // initialization
refLocal = ref anotherVeryLargeStruct; // reassigned, refLocal refers to different storage.
```

Ref local variables must still be initialized when they are declared.

Ref returns and ref locals: an example

The following example defines a `NumberStore` class that stores an array of integer values. The `FindNumber` method returns by reference the first number that is greater than or equal to the number passed as an argument. If no number is greater than or equal to the argument, the method returns the number in index 0.

```
using System;

class NumberStore
{
    int[] numbers = { 1, 3, 7, 15, 31, 63, 127, 255, 511, 1023 };

    public ref int FindNumber(int target)
    {
        for (int ctr = 0; ctr < numbers.Length; ctr++)
        {
            if (numbers[ctr] >= target)
                return ref numbers[ctr];
        }
        return ref numbers[0];
    }

    public override string ToString() => string.Join(" ", numbers);
}
```

The following example calls the `NumberStore.FindNumber` method to retrieve the first value that is greater than or equal to 16. The caller then doubles the value returned by the method. The output from the example shows the change reflected in the value of the array elements of the `NumberStore` instance.

```
var store = new NumberStore();
Console.WriteLine($"Original sequence: {store.ToString()}");
int number = 16;
ref var value = ref store.FindNumber(number);
value *= 2;
Console.WriteLine($"New sequence:      {store.ToString()}");
// The example displays the following output:
//      Original sequence: 1 3 7 15 31 63 127 255 511 1023
//      New sequence:      1 3 7 15 62 63 127 255 511 1023
```

Without support for reference return values, such an operation is performed by returning the index of the array element along with its value. The caller can then use this index to modify the value in a separate method call. However, the caller can also modify the index to access and possibly modify other array values.

The following example shows how the `FindNumber` method could be rewritten after C# 7.3 to use ref local reassignment:

```
using System;

class NumberStore
{
    int[] numbers = { 1, 3, 7, 15, 31, 63, 127, 255, 511, 1023 };

    public ref int FindNumber(int target)
    {
        ref int returnVal = ref numbers[0];
        var ctr = numbers.Length - 1;
        while ((ctr >= 0) && (numbers[ctr] >= target))
        {
            returnVal = ref numbers[ctr];
            ctr--;
        }
        return ref returnVal;
    }

    public override string ToString() => string.Join(" ", numbers);
}
```

This second version is more efficient with longer sequences in scenarios where the number sought is closer to the end of the array, as the array is iterated from end towards the beginning, causing fewer items to be examined.

See also

- [ref keyword](#)
- [Write safe efficient code](#)

Passing Parameters (C# Programming Guide)

12/28/2021 • 2 minutes to read • [Edit Online](#)

In C#, arguments can be passed to parameters either by value or by reference. Passing by reference enables function members, methods, properties, indexers, operators, and constructors to change the value of the parameters and have that change persist in the calling environment. To pass a parameter by reference with the intent of changing the value, use the `ref`, or `out` keyword. To pass by reference with the intent of avoiding copying but not changing the value, use the `in` modifier. For simplicity, only the `ref` keyword is used in the examples in this topic. For more information about the difference between `in`, `ref`, and `out`, see [in](#), [ref](#), and [out](#).

The following example illustrates the difference between value and reference parameters.

```
class Program
{
    static void Main(string[] args)
    {
        int arg;

        // Passing by value.
        // The value of arg in Main is not changed.
        arg = 4;
        squareVal(arg);
        Console.WriteLine(arg);
        // Output: 4

        // Passing by reference.
        // The value of arg in Main is changed.
        arg = 4;
        squareRef(ref arg);
        Console.WriteLine(arg);
        // Output: 16
    }

    static void squareVal(int valParameter)
    {
        valParameter *= valParameter;
    }

    // Passing by reference
    static void squareRef(ref int refParameter)
    {
        refParameter *= refParameter;
    }
}
```

For more information, see the following topics:

- [Passing Value-Type Parameters](#)
- [Passing Reference-Type Parameters](#)

C# Language Specification

For more information, see [Argument lists](#) in the [C# Language Specification](#). The language specification is the definitive source for C# syntax and usage.

See also

- [C# Programming Guide](#)
- [Methods](#)

Passing Value-Type Parameters (C# Programming Guide)

12/28/2021 • 3 minutes to read • [Edit Online](#)

A **value-type** variable contains its data directly as opposed to a **reference-type** variable, which contains a reference to its data. Passing a value-type variable to a method by value means passing a copy of the variable to the method. Any changes to the parameter that take place inside the method have no effect on the original data stored in the argument variable. If you want the called method to change the value of the argument, you must pass it by reference, using the **ref** or **out** keyword. You may also use the **in** keyword to pass a value parameter by reference to avoid the copy while guaranteeing that the value will not be changed. For simplicity, the following examples use `ref`.

Passing Value Types by Value

The following example demonstrates passing value-type parameters by value. The variable `n` is passed by value to the method `SquareIt`. Any changes that take place inside the method have no effect on the original value of the variable.

```
class PassingValByVal
{
    static void SquareIt(int x)
    // The parameter x is passed by value.
    // Changes to x will not affect the original value of x.
    {
        x *= x;
        System.Console.WriteLine("The value inside the method: {0}", x);
    }
    static void Main()
    {
        int n = 5;
        System.Console.WriteLine("The value before calling the method: {0}", n);

        SquareIt(n); // Passing the variable by value.
        System.Console.WriteLine("The value after calling the method: {0}", n);

        // Keep the console window open in debug mode.
        System.Console.WriteLine("Press any key to exit.");
        System.Console.ReadKey();
    }
}
/* Output:
The value before calling the method: 5
The value inside the method: 25
The value after calling the method: 5
*/
```

The variable `n` is a value type. It contains its data, the value `5`. When `SquareIt` is invoked, the contents of `n` are copied into the parameter `x`, which is squared inside the method. In `Main`, however, the value of `n` is the same after calling the `SquareIt` method as it was before. The change that takes place inside the method only affects the local variable `x`.

Passing Value Types by Reference

The following example is the same as the previous example, except that the argument is passed as a `ref` parameter. The value of the underlying argument, `n`, is changed when `x` is changed in the method.

```
class PassingValByRef
{
    static void SquareIt(ref int x)
    // The parameter x is passed by reference.
    // Changes to x will affect the original value of x.
    {
        x *= x;
        System.Console.WriteLine("The value inside the method: {0}", x);
    }
    static void Main()
    {
        int n = 5;
        System.Console.WriteLine("The value before calling the method: {0}", n);

        SquareIt(ref n); // Passing the variable by reference.
        System.Console.WriteLine("The value after calling the method: {0}", n);

        // Keep the console window open in debug mode.
        System.Console.WriteLine("Press any key to exit.");
        System.Console.ReadKey();
    }
}
/* Output:
The value before calling the method: 5
The value inside the method: 25
The value after calling the method: 25
*/
```

In this example, it is not the value of `n` that is passed; rather, a reference to `n` is passed. The parameter `x` is not an `int`; it is a reference to an `int`, in this case, a reference to `n`. Therefore, when `x` is squared inside the method, what actually is squared is what `x` refers to, `n`.

Swapping Value Types

A common example of changing the values of arguments is a swap method, where you pass two variables to the method, and the method swaps their contents. You must pass the arguments to the swap method by reference. Otherwise, you swap local copies of the parameters inside the method, and no change occurs in the calling method. The following example swaps integer values.

```
static void SwapByRef(ref int x, ref int y)
{
    int temp = x;
    x = y;
    y = temp;
}
```

When you call the `SwapByRef` method, use the `ref` keyword in the call, as shown in the following example.


```
static void Main()
{
    int i = 2, j = 3;
    System.Console.WriteLine("i = {0}  j = {1}" , i, j);

    SwapByRef (ref i, ref j);

    System.Console.WriteLine("i = {0}  j = {1}" , i, j);

    // Keep the console window open in debug mode.
    System.Console.WriteLine("Press any key to exit.");
    System.Console.ReadKey();
}
/* Output:
   i = 2  j = 3
   i = 3  j = 2
*/
```

See also

- [C# Programming Guide](#)
- [Passing Parameters](#)
- [Passing Reference-Type Parameters](#)

Passing Reference-Type Parameters (C# Programming Guide)

12/28/2021 • 4 minutes to read • [Edit Online](#)

A variable of a [reference type](#) does not contain its data directly; it contains a reference to its data. When you pass a reference-type parameter by value, it is possible to change the data belonging to the referenced object, such as the value of a class member. However, you cannot change the value of the reference itself; for example, you cannot use the same reference to allocate memory for a new object and have it persist outside the method. To do that, pass the parameter using the [ref](#) or [out](#) keyword. For simplicity, the following examples use [ref](#).

Passing Reference Types by Value

The following example demonstrates passing a reference-type parameter, `arr`, by value, to a method, `Change`. Because the parameter is a reference to `arr`, it is possible to change the values of the array elements. However, the attempt to reassign the parameter to a different memory location only works inside the method and does not affect the original variable, `arr`.

```
class PassingRefByVal
{
    static void Change(int[] pArray)
    {
        pArray[0] = 888; // This change affects the original element.
        pArray = new int[5] {-3, -1, -2, -3, -4}; // This change is local.
        System.Console.WriteLine("Inside the method, the first element is: {0}", pArray[0]);
    }

    static void Main()
    {
        int[] arr = {1, 4, 5};
        System.Console.WriteLine("Inside Main, before calling the method, the first element is: {0}", arr
[0]);

        Change(arr);
        System.Console.WriteLine("Inside Main, after calling the method, the first element is: {0}", arr
[0]);
    }
}
/* Output:
    Inside Main, before calling the method, the first element is: 1
    Inside the method, the first element is: -3
    Inside Main, after calling the method, the first element is: 888
*/
```

In the preceding example, the array, `arr`, which is a reference type, is passed to the method without the [ref](#) parameter. In such a case, a copy of the reference, which points to `arr`, is passed to the method. The output shows that it is possible for the method to change the contents of an array element, in this case from `1` to `888`. However, allocating a new portion of memory by using the [new](#) operator inside the `Change` method makes the variable `pArray` reference a new array. Thus, any changes after that will not affect the original array, `arr`, which is created inside `Main`. In fact, two arrays are created in this example, one inside `Main` and one inside the `Change` method.

Passing Reference Types by Reference

The following example is the same as the previous example, except that the `ref` keyword is added to the method header and call. Any changes that take place in the method affect the original variable in the calling program.

```
class PassingRefByRef
{
    static void Change(ref int[] pArray)
    {
        // Both of the following changes will affect the original variables:
        pArray[0] = 888;
        pArray = new int[5] {-3, -1, -2, -3, -4};
        System.Console.WriteLine("Inside the method, the first element is: {0}", pArray[0]);
    }

    static void Main()
    {
        int[] arr = {1, 4, 5};
        System.Console.WriteLine("Inside Main, before calling the method, the first element is: {0}",
arr[0]);

        Change(ref arr);
        System.Console.WriteLine("Inside Main, after calling the method, the first element is: {0}",
arr[0]);
    }
}
/* Output:
    Inside Main, before calling the method, the first element is: 1
    Inside the method, the first element is: -3
    Inside Main, after calling the method, the first element is: -3
*/
```

All of the changes that take place inside the method affect the original array in `Main`. In fact, the original array is reallocated using the `new` operator. Thus, after calling the `Change` method, any reference to `arr` points to the five-element array, which is created in the `Change` method.

Swapping Two Strings

Swapping strings is a good example of passing reference-type parameters by reference. In the example, two strings, `str1` and `str2`, are initialized in `Main` and passed to the `SwapStrings` method as parameters modified by the `ref` keyword. The two strings are swapped inside the method and inside `Main` as well.

```

class SwappingStrings
{
    static void SwapStrings(ref string s1, ref string s2)
    // The string parameter is passed by reference.
    // Any changes on parameters will affect the original variables.
    {
        string temp = s1;
        s1 = s2;
        s2 = temp;
        System.Console.WriteLine("Inside the method: {0} {1}", s1, s2);
    }

    static void Main()
    {
        string str1 = "John";
        string str2 = "Smith";
        System.Console.WriteLine("Inside Main, before swapping: {0} {1}", str1, str2);

        SwapStrings(ref str1, ref str2);    // Passing strings by reference
        System.Console.WriteLine("Inside Main, after swapping: {0} {1}", str1, str2);
    }
}
/* Output:
    Inside Main, before swapping: John Smith
    Inside the method: Smith John
    Inside Main, after swapping: Smith John
*/

```

In this example, the parameters need to be passed by reference to affect the variables in the calling program. If you remove the `ref` keyword from both the method header and the method call, no changes will take place in the calling program.

For more information about strings, see [string](#).

See also

- [C# Programming Guide](#)
- [Passing Parameters](#)
- [ref](#)
- [in](#)
- [out](#)
- [Reference Types](#)

How to know the difference between passing a struct and passing a class reference to a method (C# Programming Guide)

12/28/2021 • 2 minutes to read • [Edit Online](#)

The following example demonstrates how passing a [struct](#) to a method differs from passing a [class](#) instance to a method. In the example, both of the arguments (struct and class instance) are passed by value, and both methods change the value of one field of the argument. However, the results of the two methods are not the same because what is passed when you pass a struct differs from what is passed when you pass an instance of a class.

Because a struct is a [value type](#), when you [pass a struct by value](#) to a method, the method receives and operates on a copy of the struct argument. The method has no access to the original struct in the calling method and therefore can't change it in any way. The method can change only the copy.

A class instance is a [reference type](#), not a value type. When [a reference type is passed by value](#) to a method, the method receives a copy of the reference to the class instance. That is, the called method receives a copy of the address of the instance, and the calling method retains the original address of the instance. The class instance in the calling method has an address, the parameter in the called method has a copy of the address, and both addresses refer to the same object. Because the parameter contains only a copy of the address, the called method cannot change the address of the class instance in the calling method. However, the called method can use the copy of the address to access the class members that both the original address and the copy of the address reference. If the called method changes a class member, the original class instance in the calling method also changes.

The output of the following example illustrates the difference. The value of the `willIChange` field of the class instance is changed by the call to method `ClassTaker` because the method uses the address in the parameter to find the specified field of the class instance. The `willIChange` field of the struct in the calling method is not changed by the call to method `StructTaker` because the value of the argument is a copy of the struct itself, not a copy of its address. `StructTaker` changes the copy, and the copy is lost when the call to `StructTaker` is completed.

Example

```

using System;

class TheClass
{
    public string willIChange;
}

struct TheStruct
{
    public string willIChange;
}

class TestClassAndStruct
{
    static void ClassTaker(TheClass c)
    {
        c.willIChange = "Changed";
    }

    static void StructTaker(TheStruct s)
    {
        s.willIChange = "Changed";
    }

    static void Main()
    {
        TheClass testClass = new TheClass();
        TheStruct testStruct = new TheStruct();

        testClass.willIChange = "Not Changed";
        testStruct.willIChange = "Not Changed";

        ClassTaker(testClass);
        StructTaker(testStruct);

        Console.WriteLine("Class field = {0}", testClass.willIChange);
        Console.WriteLine("Struct field = {0}", testStruct.willIChange);

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
/* Output:
   Class field = Changed
   Struct field = Not Changed
*/

```

See also

- [C# Programming Guide](#)
- [Classes](#)
- [Structure types](#)
- [Passing Parameters](#)

Implicitly typed local variables (C# Programming Guide)

12/28/2021 • 5 minutes to read • [Edit Online](#)

Local variables can be declared without giving an explicit type. The `var` keyword instructs the compiler to infer the type of the variable from the expression on the right side of the initialization statement. The inferred type may be a built-in type, an anonymous type, a user-defined type, or a type defined in the .NET class library. For more information about how to initialize arrays with `var`, see [Implicitly Typed Arrays](#).

The following examples show various ways in which local variables can be declared with `var`:

```
// i is compiled as an int
var i = 5;

// s is compiled as a string
var s = "Hello";

// a is compiled as int[]
var a = new[] { 0, 1, 2 };

// expr is compiled as IEnumerable<Customer>
// or perhaps IQueryable<Customer>
var expr =
    from c in customers
    where c.City == "London"
    select c;

// anon is compiled as an anonymous type
var anon = new { Name = "Terry", Age = 34 };

// list is compiled as List<int>
var list = new List<int>();
```

It is important to understand that the `var` keyword does not mean "variant" and does not indicate that the variable is loosely typed, or late-bound. It just means that the compiler determines and assigns the most appropriate type.

The `var` keyword may be used in the following contexts:

- On local variables (variables declared at method scope) as shown in the previous example.
- In a `for` initialization statement.

```
for (var x = 1; x < 10; x++)
```

- In a `foreach` initialization statement.

```
foreach (var item in list) {...}
```

- In a `using` statement.

```
using (var file = new StreamReader("C:\\myfile.txt")) {...}
```

For more information, see [How to use implicitly typed local variables and arrays in a query expression](#).

var and anonymous types

In many cases the use of `var` is optional and is just a syntactic convenience. However, when a variable is initialized with an anonymous type you must declare the variable as `var` if you need to access the properties of the object at a later point. This is a common scenario in LINQ query expressions. For more information, see [Anonymous Types](#).

From the perspective of your source code, an anonymous type has no name. Therefore, if a query variable has been initialized with `var`, then the only way to access the properties in the returned sequence of objects is to use `var` as the type of the iteration variable in the `foreach` statement.

```
class ImplicitlyTypedLocals2
{
    static void Main()
    {
        string[] words = { "aPPLE", "BlUeBeRrY", "cHeRry" };

        // If a query produces a sequence of anonymous types,
        // then use var in the foreach statement to access the properties.
        var upperLowerWords =
            from w in words
            select new { Upper = w.ToUpper(), Lower = w.ToLower() };

        // Execute the query
        foreach (var ul in upperLowerWords)
        {
            Console.WriteLine("Uppercase: {0}, Lowercase: {1}", ul.Upper, ul.Lower);
        }
    }
}

/* Outputs:
    Uppercase: APPLE, Lowercase: apple
    Uppercase: BLUEBERRY, Lowercase: blueberry
    Uppercase: CHERRY, Lowercase: cherry
*/
```

Remarks

The following restrictions apply to implicitly-typed variable declarations:

- `var` can only be used when a local variable is declared and initialized in the same statement; the variable cannot be initialized to null, or to a method group or an anonymous function.
- `var` cannot be used on fields at class scope.
- Variables declared by using `var` cannot be used in the initialization expression. In other words, this expression is legal: `int i = (i = 20);` but this expression produces a compile-time error:
`var i = (i = 20);`
- Multiple implicitly-typed variables cannot be initialized in the same statement.
- If a type named `var` is in scope, then the `var` keyword will resolve to that type name and will not be treated as part of an implicitly typed local variable declaration.

Implicit typing with the `var` keyword can only be applied to variables at local method scope. Implicit typing is not available for class fields as the C# compiler would encounter a logical paradox as it processed the code: the compiler needs to know the type of the field, but it cannot determine the type until the assignment expression is

analyzed, and the expression cannot be evaluated without knowing the type. Consider the following code:

```
private var bookTitles;
```

`bookTitles` is a class field given the type `var`. As the field has no expression to evaluate, it is impossible for the compiler to infer what type `bookTitles` is supposed to be. In addition, adding an expression to the field (like you would for a local variable) is also insufficient:

```
private var bookTitles = new List<string>();
```

When the compiler encounters fields during code compilation, it records each field's type before processing any expressions associated with it. The compiler encounters the same paradox trying to parse `bookTitles`: it needs to know the type of the field, but the compiler would normally determine `var`'s type by analyzing the expression, which isn't possible without knowing the type beforehand.

You may find that `var` can also be useful with query expressions in which the exact constructed type of the query variable is difficult to determine. This can occur with grouping and ordering operations.

The `var` keyword can also be useful when the specific type of the variable is tedious to type on the keyboard, or is obvious, or does not add to the readability of the code. One example where `var` is helpful in this manner is with nested generic types such as those used with group operations. In the following query, the type of the query variable is `IEnumerable<IGrouping<string, Student>>`. As long as you and others who must maintain your code understand this, there is no problem with using implicit typing for convenience and brevity.

```
// Same as previous example except we use the entire last name as a key.  
// Query variable is an IEnumerable<IGrouping<string, Student>>  
var studentQuery3 =  
    from student in students  
    group student by student.Last;
```

The use of `var` helps simplify your code, but its use should be restricted to cases where it is required, or when it makes your code easier to read. For more information about when to use `var` properly, see the [Implicitly typed local variables](#) section on the C# Coding Guidelines article.

See also

- [C# Reference](#)
- [Implicitly Typed Arrays](#)
- [How to use implicitly typed local variables and arrays in a query expression](#)
- [Anonymous Types](#)
- [Object and Collection Initializers](#)
- [var](#)
- [LINQ in C#](#)
- [LINQ \(Language-Integrated Query\)](#)
- [Iteration statements](#)
- [using Statement](#)

How to use implicitly typed local variables and arrays in a query expression (C# Programming Guide)

12/28/2021 • 2 minutes to read • [Edit Online](#)

You can use implicitly typed local variables whenever you want the compiler to determine the type of a local variable. You must use implicitly typed local variables to store anonymous types, which are often used in query expressions. The following examples illustrate both optional and required uses of implicitly typed local variables in queries.

Implicitly typed local variables are declared by using the `var` contextual keyword. For more information, see [Implicitly Typed Local Variables](#) and [Implicitly Typed Arrays](#).

Examples

The following example shows a common scenario in which the `var` keyword is required: a query expression that produces a sequence of anonymous types. In this scenario, both the query variable and the iteration variable in the `foreach` statement must be implicitly typed by using `var` because you do not have access to a type name for the anonymous type. For more information about anonymous types, see [Anonymous Types](#).

```
private static void QueryNames(char firstLetter)
{
    // Create the query. Use of var is required because
    // the query produces a sequence of anonymous types:
    // System.Collections.Generic.IEnumerable<??>.
    var studentQuery =
        from student in students
        where student.FirstName[0] == firstLetter
        select new { student.FirstName, student.LastName };

    // Execute the query and display the results.
    foreach (var anonType in studentQuery)
    {
        Console.WriteLine("First = {0}, Last = {1}", anonType.FirstName, anonType.LastName);
    }
}
```

The following example uses the `var` keyword in a situation that is similar, but in which the use of `var` is optional. Because `student.LastName` is a string, execution of the query returns a sequence of strings. Therefore, the type of `queryID` could be declared as `System.Collections.Generic.IEnumerable<string>` instead of `var`. Keyword `var` is used for convenience. In the example, the iteration variable in the `foreach` statement is explicitly typed as a string, but it could instead be declared by using `var`. Because the type of the iteration variable is not an anonymous type, the use of `var` is an option, not a requirement. Remember, `var` itself is not a type, but an instruction to the compiler to infer and assign the type.

```
// Variable queryId could be declared by using
// System.Collections.Generic.IEnumerable<string>
// instead of var.
var queryId =
    from student in students
    where student.Id > 111
    select student.LastName;

// Variable str could be declared by using var instead of string.
foreach (string str in queryId)
{
    Console.WriteLine("Last name: {0}", str);
}
```

See also

- [C# Programming Guide](#)
- [Extension Methods](#)
- [LINQ \(Language-Integrated Query\)](#)
- [var](#)
- [LINQ in C#](#)

Extension Methods (C# Programming Guide)

12/28/2021 • 10 minutes to read • [Edit Online](#)

Extension methods enable you to "add" methods to existing types without creating a new derived type, recompiling, or otherwise modifying the original type. Extension methods are static methods, but they're called as if they were instance methods on the extended type. For client code written in C#, F# and Visual Basic, there's no apparent difference between calling an extension method and the methods defined in a type.

The most common extension methods are the LINQ standard query operators that add query functionality to the existing [System.Collections.IEnumerable](#) and [System.Collections.Generic.IEnumerable<T>](#) types. To use the standard query operators, first bring them into scope with a `using System.Linq` directive. Then any type that implements [IEnumerable<T>](#) appears to have instance methods such as [GroupBy](#), [OrderBy](#), [Average](#), and so on. You can see these additional methods in IntelliSense statement completion when you type "dot" after an instance of an [IEnumerable<T>](#) type such as [List<T>](#) or [Array](#).

OrderBy Example

The following example shows how to call the standard query operator `OrderBy` method on an array of integers. The expression in parentheses is a lambda expression. Many standard query operators take lambda expressions as parameters, but this isn't a requirement for extension methods. For more information, see [Lambda Expressions](#).

```
class ExtensionMethods2
{
    static void Main()
    {
        int[] ints = { 10, 45, 15, 39, 21, 26 };
        var result = ints.OrderBy(g => g);
        foreach (var i in result)
        {
            System.Console.Write(i + " ");
        }
    }
}
//Output: 10 15 21 26 39 45
```

Extension methods are defined as static methods but are called by using instance method syntax. Their first parameter specifies which type the method operates on. The parameter is preceded by the [this](#) modifier. Extension methods are only in scope when you explicitly import the namespace into your source code with a `using` directive.

The following example shows an extension method defined for the [System.String](#) class. It's defined inside a non-nested, non-generic static class:

```
namespace ExtensionMethods
{
    public static class MyExtensions
    {
        public static int WordCount(this String str)
        {
            return str.Split(new char[] { ' ', '.', '?' },
                            StringSplitOptions.RemoveEmptyEntries).Length;
        }
    }
}
```

The `WordCount` extension method can be brought into scope with this `using` directive:

```
using ExtensionMethods;
```

And it can be called from an application by using this syntax:

```
string s = "Hello Extension Methods";
int i = s.WordCount();
```

You invoke the extension method in your code with instance method syntax. The intermediate language (IL) generated by the compiler translates your code into a call on the static method. The principle of encapsulation is not really being violated. Extension methods cannot access private variables in the type they are extending.

Both the `MyExtensions` class and the `WordCount` method are `static`, and it can be accessed like all other `static` members. The `WordCount` method can be invoked like other `static` methods as follows:

```
string s = "Hello Extension Methods";
int i = MyExtensions.WordCount(s);
```

The preceding C# code:

- Declares and assigns a new `string` named `s` with a value of `"Hello Extension Methods"`.
- Calls `MyExtensions.WordCount` given argument `s`

For more information, see [How to implement and call a custom extension method](#).

In general, you'll probably be calling extension methods far more often than implementing your own. Because extension methods are called by using instance method syntax, no special knowledge is required to use them from client code. To enable extension methods for a particular type, just add a `using` directive for the namespace in which the methods are defined. For example, to use the standard query operators, add this `using` directive to your code:

```
using System.Linq;
```

(You may also have to add a reference to `System.Core.dll`.) You'll notice that the standard query operators now appear in IntelliSense as additional methods available for most `IEnumerable<T>` types.

Binding Extension Methods at Compile Time

You can use extension methods to extend a class or interface, but not to override them. An extension method with the same name and signature as an interface or class method will never be called. At compile time, extension methods always have lower priority than instance methods defined in the type itself. In other words, if

a type has a method named `Process(int i)`, and you have an extension method with the same signature, the compiler will always bind to the instance method. When the compiler encounters a method invocation, it first looks for a match in the type's instance methods. If no match is found, it will search for any extension methods that are defined for the type, and bind to the first extension method that it finds. The following example demonstrates how the compiler determines which extension method or instance method to bind to.

Example

The following example demonstrates the rules that the C# compiler follows in determining whether to bind a method call to an instance method on the type, or to an extension method. The static class `Extensions` contains extension methods defined for any type that implements `IMyInterface`. Classes `A`, `B`, and `C` all implement the interface.

The `MethodB` extension method is never called because its name and signature exactly match methods already implemented by the classes.

When the compiler can't find an instance method with a matching signature, it will bind to a matching extension method if one exists.

```
// Define an interface named IMyInterface.
namespace DefineIMyInterface
{
    using System;

    public interface IMyInterface
    {
        // Any class that implements IMyInterface must define a method
        // that matches the following signature.
        void MethodB();
    }
}

// Define extension methods for IMyInterface.
namespace Extensions
{
    using System;
    using DefineIMyInterface;

    // The following extension methods can be accessed by instances of any
    // class that implements IMyInterface.
    public static class Extension
    {
        public static void MethodA(this IMyInterface myInterface, int i)
        {
            Console.WriteLine
                ("Extension.MethodA(this IMyInterface myInterface, int i)");
        }

        public static void MethodA(this IMyInterface myInterface, string s)
        {
            Console.WriteLine
                ("Extension.MethodA(this IMyInterface myInterface, string s)");
        }

        // This method is never called in ExtensionMethodsDemo1, because each
        // of the three classes A, B, and C implements a method named MethodB
        // that has a matching signature.
        public static void MethodB(this IMyInterface myInterface)
        {
            Console.WriteLine
                ("Extension.MethodB(this IMyInterface myInterface)");
        }
    }
}
```

```

// Define three classes that implement IMyInterface, and then use them to test
// the extension methods.
namespace ExtensionMethodsDemo1
{
    using System;
    using Extensions;
    using DefineIMyInterface;

    class A : IMyInterface
    {
        public void MethodB() { Console.WriteLine("A.MethodB()"); }
    }

    class B : IMyInterface
    {
        public void MethodB() { Console.WriteLine("B.MethodB()"); }
        public void MethodA(int i) { Console.WriteLine("B.MethodA(int i)"); }
    }

    class C : IMyInterface
    {
        public void MethodB() { Console.WriteLine("C.MethodB()"); }
        public void MethodA(object obj)
        {
            Console.WriteLine("C.MethodA(object obj)");
        }
    }

    class ExtMethodDemo
    {
        static void Main(string[] args)
        {
            // Declare an instance of class A, class B, and class C.
            A a = new A();
            B b = new B();
            C c = new C();

            // For a, b, and c, call the following methods:
            //      -- MethodA with an int argument
            //      -- MethodA with a string argument
            //      -- MethodB with no argument.

            // A contains no MethodA, so each call to MethodA resolves to
            // the extension method that has a matching signature.
            a.MethodA(1);           // Extension.MethodA(IMyInterface, int)
            a.MethodA("hello");     // Extension.MethodA(IMyInterface, string)

            // A has a method that matches the signature of the following call
            // to MethodB.
            a.MethodB();           // A.MethodB()

            // B has methods that match the signatures of the following
            // method calls.
            b.MethodA(1);          // B.MethodA(int)
            b.MethodB();           // B.MethodB()

            // B has no matching method for the following call, but
            // class Extension does.
            b.MethodA("hello");    // Extension.MethodA(IMyInterface, string)

            // C contains an instance method that matches each of the following
            // method calls.
            c.MethodA(1);          // C.MethodA(object)
            c.MethodA("hello");    // C.MethodA(object)
            c.MethodB();           // C.MethodB()
        }
    }
}

```

```

s
/* Output:
    Extension.MethodA(this IMyInterface myInterface, int i)
    Extension.MethodA(this IMyInterface myInterface, string s)
    A.MethodB()
    B.MethodA(int i)
    B.MethodB()
    Extension.MethodA(this IMyInterface myInterface, string s)
    C.MethodA(object obj)
    C.MethodA(object obj)
    C.MethodB()
*/

```

Common Usage Patterns

Collection Functionality

In the past, it was common to create "Collection Classes" that implemented the [System.Collections.Generic.IEnumerable<T>](#) interface for a given type and contained functionality that acted on collections of that type. While there's nothing wrong with creating this type of collection object, the same functionality can be achieved by using an extension on the [System.Collections.Generic.IEnumerable<T>](#). Extensions have the advantage of allowing the functionality to be called from any collection such as an [System.Array](#) or [System.Collections.Generic.List<T>](#) that implements [System.Collections.Generic.IEnumerable<T>](#) on that type. An example of this using an Array of Int32 can be found [earlier in this article](#).

Layer-Specific Functionality

When using an Onion Architecture or other layered application design, it's common to have a set of Domain Entities or Data Transfer Objects that can be used to communicate across application boundaries. These objects generally contain no functionality, or only minimal functionality that applies to all layers of the application. Extension methods can be used to add functionality that is specific to each application layer without loading the object down with methods not needed or wanted in other layers.

```

public class DomainEntity
{
    public int Id { get; set; }
    public string FirstName { get; set; }
    public string LastName { get; set; }
}

static class DomainEntityExtensions
{
    static string FullName(this DomainEntity value)
        => $"{value.FirstName} {value.LastName}";
}

```

Extending Predefined Types

Rather than creating new objects when reusable functionality needs to be created, we can often extend an existing type, such as a .NET or CLR type. As an example, if we don't use extension methods, we might create an `Engine` or `Query` class to do the work of executing a query on a SQL Server that may be called from multiple places in our code. However we can instead extend the [System.Data.SqlClient.SqlConnection](#) class using extension methods to perform that query from anywhere we have a connection to a SQL Server. Other examples might be to add common functionality to the [System.String](#) class, extend the data processing capabilities of the [System.IO.File](#) and [System.IO.Stream](#) objects, and [System.Exception](#) objects for specific error handling functionality. These types of use-cases are limited only by your imagination and good sense.

Extending predefined types can be difficult with `struct` types because they're passed by value to methods. That means any changes to the struct are made to a copy of the struct. Those changes aren't visible once the

extension method exists. Beginning with C# 7.2, you can add the `ref` modifier to the first argument of an extension method. Adding the `ref` modifier means the first argument is passed by reference. This enables you to write extension methods that change the state of the struct being extended.

General Guidelines

While it's still considered preferable to add functionality by modifying an object's code or deriving a new type whenever it's reasonable and possible to do so, extension methods have become a crucial option for creating reusable functionality throughout the .NET ecosystem. For those occasions when the original source isn't under your control, when a derived object is inappropriate or impossible, or when the functionality shouldn't be exposed beyond its applicable scope, Extension methods are an excellent choice.

For more information on derived types, see [Inheritance](#).

When using an extension method to extend a type whose source code you aren't in control of, you run the risk that a change in the implementation of the type will cause your extension method to break.

If you do implement extension methods for a given type, remember the following points:

- An extension method will never be called if it has the same signature as a method defined in the type.
- Extension methods are brought into scope at the namespace level. For example, if you have multiple static classes that contain extension methods in a single namespace named `Extensions`, they'll all be brought into scope by the `using Extensions;` directive.

For a class library that you implemented, you shouldn't use extension methods to avoid incrementing the version number of an assembly. If you want to add significant functionality to a library for which you own the source code, follow the .NET guidelines for assembly versioning. For more information, see [Assembly Versioning](#).

See also

- [C# Programming Guide](#)
- [Parallel Programming Samples \(these include many example extension methods\)](#)
- [Lambda Expressions](#)
- [Standard Query Operators Overview](#)
- [Conversion rules for Instance parameters and their impact](#)
- [Extension methods Interoperability between languages](#)
- [Extension methods and Curried Delegates](#)
- [Extension method Binding and Error reporting](#)

How to implement and call a custom extension method (C# Programming Guide)

12/28/2021 • 2 minutes to read • [Edit Online](#)

This topic shows how to implement your own extension methods for any .NET type. Client code can use your extension methods by adding a reference to the DLL that contains them, and adding a [using](#) directive that specifies the namespace in which the extension methods are defined.

To define and call the extension method

1. Define a static [class](#) to contain the extension method.

The class must be visible to client code. For more information about accessibility rules, see [Access Modifiers](#).

2. Implement the extension method as a static method with at least the same visibility as the containing class.
3. The first parameter of the method specifies the type that the method operates on; it must be preceded with the [this](#) modifier.
4. In the calling code, add a `using` directive to specify the [namespace](#) that contains the extension method class.
5. Call the methods as if they were instance methods on the type.

Note that the first parameter is not specified by calling code because it represents the type on which the operator is being applied, and the compiler already knows the type of your object. You only have to provide arguments for parameters 2 through `n`.

Example

The following example implements an extension method named `WordCount` in the `CustomExtensions.StringExtension` class. The method operates on the [String](#) class, which is specified as the first method parameter. The `CustomExtensions` namespace is imported into the application namespace, and the method is called inside the `Main` method.

```

using System.Linq;
using System.Text;
using System;

namespace CustomExtensions
{
    // Extension methods must be defined in a static class.
    public static class StringExtension
    {
        // This is the extension method.
        // The first parameter takes the "this" modifier
        // and specifies the type for which the method is defined.
        public static int WordCount(this String str)
        {
            return str.Split(new char[] { ' ', '.', '?' }, StringSplitOptions.RemoveEmptyEntries).Length;
        }
    }
}

namespace Extension_Methods_Simple
{
    // Import the extension method namespace.
    using CustomExtensions;
    class Program
    {
        static void Main(string[] args)
        {
            string s = "The quick brown fox jumped over the lazy dog.";
            // Call the method as if it were an
            // instance method on the type. Note that the first
            // parameter is not specified by the calling code.
            int i = s.WordCount();
            System.Console.WriteLine("Word count of s is {0}", i);
        }
    }
}

```

.NET Security

Extension methods present no specific security vulnerabilities. They can never be used to impersonate existing methods on a type, because all name collisions are resolved in favor of the instance or static method defined by the type itself. Extension methods cannot access any private data in the extended class.

See also

- [C# Programming Guide](#)
- [Extension Methods](#)
- [LINQ \(Language-Integrated Query\)](#)
- [Static Classes and Static Class Members](#)
- [protected](#)
- [internal](#)
- [public](#)
- [this](#)
- [namespace](#)

How to create a new method for an enumeration (C# Programming Guide)

12/28/2021 • 2 minutes to read • [Edit Online](#)

You can use extension methods to add functionality specific to a particular enum type.

Example

In the following example, the `Grades` enumeration represents the possible letter grades that a student may receive in a class. An extension method named `Passing` is added to the `Grades` type so that each instance of that type now "knows" whether it represents a passing grade or not.

```
using System;
using System.Collections.Generic;
using System.Text;
using System.Linq;

namespace EnumExtension
{
    // Define an extension method in a non-nested static class.
    public static class Extensions
    {
        public static Grades minPassing = Grades.D;
        public static bool Passing(this Grades grade)
        {
            return grade >= minPassing;
        }
    }

    public enum Grades { F = 0, D=1, C=2, B=3, A=4 };
    class Program
    {
        static void Main(string[] args)
        {
            Grades g1 = Grades.D;
            Grades g2 = Grades.F;
            Console.WriteLine("First {0} a passing grade.", g1.Passing() ? "is" : "is not");
            Console.WriteLine("Second {0} a passing grade.", g2.Passing() ? "is" : "is not");

            Extensions.minPassing = Grades.C;
            Console.WriteLine("\r\nRaising the bar!\r\n");
            Console.WriteLine("First {0} a passing grade.", g1.Passing() ? "is" : "is not");
            Console.WriteLine("Second {0} a passing grade.", g2.Passing() ? "is" : "is not");
        }
    }
}

/* Output:
First is a passing grade.
Second is not a passing grade.

Raising the bar!

First is not a passing grade.
Second is not a passing grade.
*/
```

Note that the `Extensions` class also contains a static variable that is updated dynamically and that the return

value of the extension method reflects the current value of that variable. This demonstrates that, behind the scenes, extension methods are invoked directly on the static class in which they are defined.

See also

- [C# Programming Guide](#)
- [Extension Methods](#)

Named and Optional Arguments (C# Programming Guide)

12/28/2021 • 7 minutes to read • [Edit Online](#)

C# 4 introduces named and optional arguments. *Named arguments* enable you to specify an argument for a parameter by matching the argument with its name rather than with its position in the parameter list. *Optional arguments* enable you to omit arguments for some parameters. Both techniques can be used with methods, indexers, constructors, and delegates.

When you use named and optional arguments, the arguments are evaluated in the order in which they appear in the argument list, not the parameter list.

Named and optional parameters enable you to supply arguments for selected parameters. This capability greatly eases calls to COM interfaces such as the Microsoft Office Automation APIs.

Named arguments

Named arguments free you from matching the order of parameters in the parameter lists of called methods. The parameter for each argument can be specified by parameter name. For example, a function that prints order details (such as, seller name, order number & product name) can be called by sending arguments by position, in the order defined by the function.

```
PrintOrderDetails("Gift Shop", 31, "Red Mug");
```

If you don't remember the order of the parameters but know their names, you can send the arguments in any order.

```
PrintOrderDetails(orderNum: 31, productName: "Red Mug", sellerName: "Gift Shop");  
PrintOrderDetails(productName: "Red Mug", sellerName: "Gift Shop", orderNum: 31);
```

Named arguments also improve the readability of your code by identifying what each argument represents. In the example method below, the `sellerName` can't be null or white space. As both `sellerName` and `productName` are string types, instead of sending arguments by position, it makes sense to use named arguments to disambiguate the two and reduce confusion for anyone reading the code.

Named arguments, when used with positional arguments, are valid as long as

- they're not followed by any positional arguments, or

```
PrintOrderDetails("Gift Shop", 31, productName: "Red Mug");
```

- *starting with C# 7.2*, they're used in the correct position. In the example below, the parameter `orderNum` is in the correct position but isn't explicitly named.

```
PrintOrderDetails(sellerName: "Gift Shop", 31, productName: "Red Mug");
```

Positional arguments that follow any out-of-order named arguments are invalid.

```
// This generates CS1738: Named argument specifications must appear after all fixed arguments have been
specified.
PrintOrderDetails(productName: "Red Mug", 31, "Gift Shop");
```

Example

The following code implements the examples from this section along with some additional ones.

```
class NamedExample
{
    static void Main(string[] args)
    {
        // The method can be called in the normal way, by using positional arguments.
        PrintOrderDetails("Gift Shop", 31, "Red Mug");

        // Named arguments can be supplied for the parameters in any order.
        PrintOrderDetails(orderNum: 31, productName: "Red Mug", sellerName: "Gift Shop");
        PrintOrderDetails(productName: "Red Mug", sellerName: "Gift Shop", orderNum: 31);

        // Named arguments mixed with positional arguments are valid
        // as long as they are used in their correct position.
        PrintOrderDetails("Gift Shop", 31, productName: "Red Mug");
        PrintOrderDetails(sellerName: "Gift Shop", 31, productName: "Red Mug");    // C# 7.2 onwards
        PrintOrderDetails("Gift Shop", orderNum: 31, "Red Mug");                // C# 7.2 onwards

        // However, mixed arguments are invalid if used out-of-order.
        // The following statements will cause a compiler error.
        // PrintOrderDetails(productName: "Red Mug", 31, "Gift Shop");
        // PrintOrderDetails(31, sellerName: "Gift Shop", "Red Mug");
        // PrintOrderDetails(31, "Red Mug", sellerName: "Gift Shop");
    }

    static void PrintOrderDetails(string sellerName, int orderNum, string productName)
    {
        if (string.IsNullOrEmpty(sellerName))
        {
            throw new ArgumentException(message: "Seller name cannot be null or empty.", paramName:
nameof(sellerName));
        }

        Console.WriteLine($"Seller: {sellerName}, Order #: {orderNum}, Product: {productName}");
    }
}
```

Optional arguments

The definition of a method, constructor, indexer, or delegate can specify its parameters are required or optional. Any call must provide arguments for all required parameters, but can omit arguments for optional parameters.

Each optional parameter has a default value as part of its definition. If no argument is sent for that parameter, the default value is used. A default value must be one of the following types of expressions:

- a constant expression;
- an expression of the form `new ValType()`, where `ValType` is a value type, such as an [enum](#) or a [struct](#);
- an expression of the form `default(ValType)`, where `ValType` is a value type.

Optional parameters are defined at the end of the parameter list, after any required parameters. If the caller provides an argument for any one of a succession of optional parameters, it must provide arguments for all preceding optional parameters. Comma-separated gaps in the argument list aren't supported. For example, in the following code, instance method `ExampleMethod` is defined with one required and two optional parameters.

```
public void ExampleMethod(int required, string optionalstr = "default string",
    int optionalint = 10)
```

The following call to `ExampleMethod` causes a compiler error, because an argument is provided for the third parameter but not for the second.

```
//anExample.ExampleMethod(3, ,4);
```

However, if you know the name of the third parameter, you can use a named argument to accomplish the task.

```
anExample.ExampleMethod(3, optionalint: 4);
```

IntelliSense uses brackets to indicate optional parameters, as shown in the following illustration:

```
anExample.ExampleMethod(  
    void ExampleClass.ExampleMethod(int required,  
        [string optionalstr = "default string"],  
        [int optionalint = 10])
```

NOTE

You can also declare optional parameters by using the .NET [OptionalAttribute](#) class. `OptionalAttribute` parameters do not require a default value.

Example

In the following example, the constructor for `ExampleClass` has one parameter, which is optional. Instance method `ExampleMethod` has one required parameter, `required`, and two optional parameters, `optionalstr` and `optionalint`. The code in `Main` shows the different ways in which the constructor and method can be invoked.

```
namespace OptionalNamespace
{
    class OptionalExample
    {
        static void Main(string[] args)
        {
            // Instance anExample does not send an argument for the constructor's
            // optional parameter.
            ExampleClass anExample = new ExampleClass();
            anExample.ExampleMethod(1, "One", 1);
            anExample.ExampleMethod(2, "Two");
            anExample.ExampleMethod(3);

            // Instance anotherExample sends an argument for the constructor's
            // optional parameter.
            ExampleClass anotherExample = new ExampleClass("Provided name");
            anotherExample.ExampleMethod(1, "One", 1);
            anotherExample.ExampleMethod(2, "Two");
            anotherExample.ExampleMethod(3);

            // The following statements produce compiler errors.

            // An argument must be supplied for the first parameter, and it
            // must be an integer.
            //anExample.ExampleMethod("One", 1);
            //anExample.ExampleMethod();

            // You cannot leave a gap in the provided arguments.
            //anExample.ExampleMethod(3, ,4);
```



```

        //anExample.ExampleMethod(3, 4);

        // You can use a named parameter to make the previous
        // statement work.
        anExample.ExampleMethod(3, optionalint: 4);
    }
}

class ExampleClass
{
    private string _name;

    // Because the parameter for the constructor, name, has a default
    // value assigned to it, it is optional.
    public ExampleClass(string name = "Default name")
    {
        _name = name;
    }

    // The first parameter, required, has no default value assigned
    // to it. Therefore, it is not optional. Both optionalstr and
    // optionalint have default values assigned to them. They are optional.
    public void ExampleMethod(int required, string optionalstr = "default string",
        int optionalint = 10)
    {
        Console.WriteLine(
            $"{_name}: {required}, {optionalstr}, and {optionalint}.");
    }
}

// The output from this example is the following:
// Default name: 1, One, and 1.
// Default name: 2, Two, and 10.
// Default name: 3, default string, and 10.
// Provided name: 1, One, and 1.
// Provided name: 2, Two, and 10.
// Provided name: 3, default string, and 10.
// Default name: 3, default string, and 4.
}

```

The preceding code shows a number of examples where optional parameters aren't applied correctly. The first illustrates that an argument must be supplied for the first parameter, which is required.

COM interfaces

Named and optional arguments, along with support for dynamic objects, greatly improve interoperability with COM APIs, such as Office Automation APIs.

For example, the [AutoFormat](#) method in the Microsoft Office Excel [Range](#) interface has seven parameters, all of which are optional. These parameters are shown in the following illustration:

```

excelApp.get_Range("A1", "B4").AutoFormat(
    dynamic Range.AutoFormat([Excel.XlRangeAutoFormat Format = 1],
        [object Number = Type.Missing], [object Font = Type.Missing],
        [object Alignment = Type.Missing], [object Border = Type.Missing],
        [object Pattern = Type.Missing], [object Width = Type.Missing])

```

In C# 3.0 and earlier versions, an argument is required for each parameter, as shown in the following example.

```
// In C# 3.0 and earlier versions, you need to supply an argument for
// every parameter. The following call specifies a value for the first
// parameter, and sends a placeholder value for the other six. The
// default values are used for those parameters.
var excelApp = new Microsoft.Office.Interop.Excel.Application();
excelApp.Workbooks.Add();
excelApp.Visible = true;

var myFormat =
    Microsoft.Office.Interop.Excel.XlRangeAutoFormat.xlRangeAutoFormatAccounting1;

excelApp.get_Range("A1", "B4").AutoFormat(myFormat, Type.Missing,
    Type.Missing, Type.Missing, Type.Missing, Type.Missing, Type.Missing);
```

However, you can greatly simplify the call to `AutoFormat` by using named and optional arguments, introduced in C# 4.0. Named and optional arguments enable you to omit the argument for an optional parameter if you don't want to change the parameter's default value. In the following call, a value is specified for only one of the seven parameters.

```
// The following code shows the same call to AutoFormat in C# 4.0. Only
// the argument for which you want to provide a specific value is listed.
excelApp.Range["A1", "B4"].AutoFormat( Format: myFormat );
```

For more information and examples, see [How to use named and optional arguments in Office programming](#) and [How to access Office interop objects by using C# features](#).

Overload resolution

Use of named and optional arguments affects overload resolution in the following ways:

- A method, indexer, or constructor is a candidate for execution if each of its parameters either is optional or corresponds, by name or by position, to a single argument in the calling statement, and that argument can be converted to the type of the parameter.
- If more than one candidate is found, overload resolution rules for preferred conversions are applied to the arguments that are explicitly specified. Omitted arguments for optional parameters are ignored.
- If two candidates are judged to be equally good, preference goes to a candidate that doesn't have optional parameters for which arguments were omitted in the call. Overload resolution generally prefers candidates that have fewer parameters.

C# language specification

For more information, see the [C# Language Specification](#). The language specification is the definitive source for C# syntax and usage.

How to use named and optional arguments in Office programming (C# Programming Guide)

12/28/2021 • 5 minutes to read • [Edit Online](#)

Named arguments and optional arguments, introduced in C# 4, enhance convenience, flexibility, and readability in C# programming. In addition, these features greatly facilitate access to COM interfaces such as the Microsoft Office automation APIs.

In the following example, method [ConvertToTable](#) has sixteen parameters that represent characteristics of a table, such as number of columns and rows, formatting, borders, fonts, and colors. All sixteen parameters are optional, because most of the time you do not want to specify particular values for all of them. However, without named and optional arguments, a value or a placeholder value has to be provided for each parameter. With named and optional arguments, you specify values only for the parameters that are required for your project.

You must have Microsoft Office Word installed on your computer to complete these procedures.

NOTE

Your computer might show different names or locations for some of the Visual Studio user interface elements in the following instructions. The Visual Studio edition that you have and the settings that you use determine these elements. For more information, see [Personalizing the IDE](#).

To create a new console application

1. Start Visual Studio.
2. On the **File** menu, point to **New**, and then click **Project**.
3. In the **Templates Categories** pane, expand **Visual C#**, and then click **Windows**.
4. Look in the top of the **Templates** pane to make sure that **.NET Framework 4** appears in the **Target Framework** box.
5. In the **Templates** pane, click **Console Application**.
6. Type a name for your project in the **Name** field.
7. Click **OK**.

The new project appears in **Solution Explorer**.

To add a reference

1. In **Solution Explorer**, right-click your IDE project's name and then click **Add Reference**. The **Add Reference** dialog box appears.
2. On the **.NET** page, select **Microsoft.Office.Interop.Word** in the **Component Name** list.
3. Click **OK**.

To add necessary using directives

1. In **Solution Explorer**, right-click the *Program.cs* file and then click **View Code**.

2. Add the following `using` directives to the top of the code file:

```
using Word = Microsoft.Office.Interop.Word;
```

To display text in a Word document

1. In the `Program` class in *Program.cs*, add the following method to create a Word application and a Word document. The `Add` method has four optional parameters. This example uses their default values. Therefore, no arguments are necessary in the calling statement.

```
static void DisplayInWord()
{
    var wordApp = new Word.Application();
    wordApp.Visible = true;
    // docs is a collection of all the Document objects currently
    // open in Word.
    Word.Documents docs = wordApp.Documents;

    // Add a document to the collection and name it doc.
    Word.Document doc = docs.Add();
}
```

2. Add the following code at the end of the method to define where to display text in the document, and what text to display:

```
// Define a range, a contiguous area in the document, by specifying
// a starting and ending character position. Currently, the document
// is empty.
Word.Range range = doc.Range(0, 0);

// Use the InsertAfter method to insert a string at the end of the
// current range.
range.InsertAfter("Testing, testing, testing. . .");
```

To run the application

1. Add the following statement to `Main`:

```
DisplayInWord();
```

2. Press CTRL+F5 to run the project. A Word document appears that contains the specified text.

To change the text to a table

1. Use the `ConvertToTable` method to enclose the text in a table. The method has sixteen optional parameters. IntelliSense encloses optional parameters in brackets, as shown in the following illustration.

```
range.ConvertToTable(|
Word.Table Range.ConvertToTable([ref object Separator = Type.Missing], [ref object NumRows =
Type.Missing], [ref object NumColumns = Type.Missing], [ref object InitialColumnWidth =
Type.Missing], [ref object Format = Type.Missing], [ref object ApplyBorders = Type.Missing],
[ref object ApplyShading = Type.Missing], [ref object ApplyFont = Type.Missing], [ref object
ApplyColor = Type.Missing], [ref object ApplyHeadingsRows = Type.Missing], [ref object
ApplyLastRow = Type.Missing], [ref object ApplyFirstColumn = Type.Missing], [ref object
ApplyLastColumn = Type.Missing], [ref object AutoFit = Type.Missing], [ref object
AutoFitBehavior = Type.Missing], [ref object DefaultTableBehavior = Type.Missing])
```

Named and optional arguments enable you to specify values for only the parameters that you want to change. Add the following code to the end of method `DisplayInWord` to create a simple table. The argument specifies that the commas in the text string in `range` separate the cells of the table.

```
// Convert to a simple table. The table will have a single row with
// three columns.
range.ConvertToTable(Separator: ",");
```

In earlier versions of C#, the call to `ConvertToTable` requires a reference argument for each parameter, as shown in the following code:

```
// Call to ConvertToTable in Visual C# 2008 or earlier. This code
// is not part of the solution.
var missing = Type.Missing;
object separator = ",";
range.ConvertToTable(ref separator, ref missing, ref missing,
    ref missing, ref missing, ref missing, ref missing,
    ref missing, ref missing, ref missing, ref missing,
    ref missing);
```

2. Press CTRL+F5 to run the project.

To experiment with other parameters

1. To change the table so that it has one column and three rows, replace the last line in `DisplayInWord` with the following statement and then type CTRL+F5.

```
range.ConvertToTable(Separator: ",", AutoFit: true, NumColumns: 1);
```

2. To specify a predefined format for the table, replace the last line in `DisplayInWord` with the following statement and then type CTRL+F5. The format can be any of the [WdTableFormat](#) constants.

```
range.ConvertToTable(Separator: ",", AutoFit: true, NumColumns: 1,
    Format: Word.WdTableFormat.wdTableFormatElegant);
```

Example

The following code includes the full example:

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using Word = Microsoft.Office.Interop.Word;

namespace OfficeHowTo
{
    class WordProgram
    {
        static void Main(string[] args)
        {
            DisplayInWord();
        }

        static void DisplayInWord()
        {
            var wordApp = new Word.Application();
            wordApp.Visible = true;
            // docs is a collection of all the Document objects currently
            // open in Word.
            Word.Documents docs = wordApp.Documents;

            // Add a document to the collection and name it doc.
            Word.Document doc = docs.Add();

            // Define a range, a contiguous area in the document, by specifying
            // a starting and ending character position. Currently, the document
            // is empty.
            Word.Range range = doc.Range(0, 0);

            // Use the InsertAfter method to insert a string at the end of the
            // current range.
            range.InsertAfter("Testing, testing, testing. .");

            // You can comment out any or all of the following statements to
            // see the effect of each one in the Word document.

            // Next, use the ConvertToTable method to put the text into a table.
            // The method has 16 optional parameters. You only have to specify
            // values for those you want to change.

            // Convert to a simple table. The table will have a single row with
            // three columns.
            range.ConvertToTable(Separator: ",");

            // Change to a single column with three rows..
            range.ConvertToTable(Separator: ",", AutoFit: true, NumColumns: 1);

            // Format the table.
            range.ConvertToTable(Separator: ",", AutoFit: true, NumColumns: 1,
                                Format: Word.WdTableFormat.wdTableFormatElegant);
        }
    }
}

```

See also

- [Named and Optional Arguments](#)

Constructors (C# programming guide)

12/28/2021 • 2 minutes to read • [Edit Online](#)

Whenever a [class](#) or [struct](#) is created, its constructor is called. A class or struct may have multiple constructors that take different arguments. Constructors enable the programmer to set default values, limit instantiation, and write code that is flexible and easy to read. For more information and examples, see [Instance constructors](#) and [Using constructors](#).

Constructor syntax

A constructor is a method whose name is the same as the name of its type. Its method signature includes only the method name and its parameter list; it does not include a return type. The following example shows the constructor for a class named `Person`.

```
public class Person
{
    private string last;
    private string first;

    public Person(string lastName, string firstName)
    {
        last = lastName;
        first = firstName;
    }

    // Remaining implementation of Person class.
}
```

If a constructor can be implemented as a single statement, you can use an [expression body definition](#). The following example defines a `Location` class whose constructor has a single string parameter named *name*. The expression body definition assigns the argument to the `locationName` field.

```
public class Location
{
    private string locationName;

    public Location(string name) => Name = name;

    public string Name
    {
        get => locationName;
        set => locationName = value;
    }
}
```

Static constructors

The previous examples have all shown instance constructors, which create a new object. A class or struct can also have a static constructor, which initializes static members of the type. Static constructors are parameterless. If you don't provide a static constructor to initialize static fields, the C# compiler initializes static fields to their default value as listed in the [Default values of C# types](#) article.

The following example uses a static constructor to initialize a static field.

```

public class Adult : Person
{
    private static int minimumAge;

    public Adult(string lastName, string firstName) : base(lastName, firstName)
    { }

    static Adult()
    {
        minimumAge = 18;
    }

    // Remaining implementation of Adult class.
}

```

You can also define a static constructor with an expression body definition, as the following example shows.

```

public class Child : Person
{
    private static int maximumAge;

    public Child(string lastName, string firstName) : base(lastName, firstName)
    { }

    static Child() => maximumAge = 18;

    // Remaining implementation of Child class.
}

```

For more information and examples, see [Static Constructors](#).

In This Section

[Using Constructors](#)

[Instance Constructors](#)

[Private Constructors](#)

[Static Constructors](#)

[How to write a copy constructor](#)

See also

- [C# Programming Guide](#)
- [The C# type system](#)
- [Finalizers](#)
- [static](#)
- [Why Do Initializers Run In The Opposite Order As Constructors? Part One](#)

Using Constructors (C# Programming Guide)

12/28/2021 • 4 minutes to read • [Edit Online](#)

When a [class](#) or [struct](#) is created, its constructor is called. Constructors have the same name as the class or struct, and they usually initialize the data members of the new object.

In the following example, a class named `Taxi` is defined by using a simple constructor. This class is then instantiated with the [new](#) operator. The `Taxi` constructor is invoked by the `new` operator immediately after memory is allocated for the new object.

```
public class Taxi
{
    public bool IsInitialized;

    public Taxi()
    {
        IsInitialized = true;
    }
}

class TestTaxi
{
    static void Main()
    {
        Taxi t = new Taxi();
        Console.WriteLine(t.IsInitialized);
    }
}
```

A constructor that takes no parameters is called a *parameterless constructor*. Parameterless constructors are invoked whenever an object is instantiated by using the `new` operator and no arguments are provided to `new`. For more information, see [Instance Constructors](#).

Unless the class is [static](#), classes without constructors are given a public parameterless constructor by the C# compiler in order to enable class instantiation. For more information, see [Static Classes and Static Class Members](#).

You can prevent a class from being instantiated by making the constructor private, as follows:

```
class NLog
{
    // Private Constructor:
    private NLog() { }

    public static double e = Math.E; //2.71828...
}
```

For more information, see [Private Constructors](#).

Constructors for [struct](#) types resemble class constructors, but `structs` cannot contain an explicit parameterless constructor because one is provided automatically by the compiler. This constructor initializes each field in the `struct` to the [default value](#). However, this parameterless constructor is only invoked if the `struct` is instantiated with `new`. For example, this code uses the parameterless constructor for [Int32](#), so that you are assured that the integer is initialized:

```
int i = new int();
Console.WriteLine(i);
```

NOTE

Beginning with C# 10, a structure type can contain an explicit parameterless constructor. For more information, see the [Parameterless constructors and field initializers](#) section of the [Structure types](#) article.

The following code, however, causes a compiler error because it does not use `new`, and because it tries to use an object that has not been initialized:

```
int i;
Console.WriteLine(i);
```

Alternatively, objects based on `structs` (including all built-in numeric types) can be initialized or assigned and then used as in the following example:

```
int a = 44; // Initialize the value type...
int b;
b = 33;     // Or assign it before using it.
Console.WriteLine("{0}, {1}", a, b);
```

So calling the parameterless constructor for a value type is not required.

Both classes and `structs` can define constructors that take parameters. Constructors that take parameters must be called through a `new` statement or a `base` statement. Classes and `structs` can also define multiple constructors, and neither is required to define a parameterless constructor. For example:

```
public class Employee
{
    public int Salary;

    public Employee() { }

    public Employee(int annualSalary)
    {
        Salary = annualSalary;
    }

    public Employee(int weeklySalary, int numberOfWeeks)
    {
        Salary = weeklySalary * numberOfWeeks;
    }
}
```

This class can be created by using either of the following statements:

```
Employee e1 = new Employee(30000);
Employee e2 = new Employee(500, 52);
```

A constructor can use the `base` keyword to call the constructor of a base class. For example:

```
public class Manager : Employee
{
    public Manager(int annualSalary)
        : base(annualSalary)
    {
        //Add further instructions here.
    }
}
```

In this example, the constructor for the base class is called before the block for the constructor is executed. The `base` keyword can be used with or without parameters. Any parameters to the constructor can be used as parameters to `base`, or as part of an expression. For more information, see [base](#).

In a derived class, if a base-class constructor is not called explicitly by using the `base` keyword, the parameterless constructor, if there is one, is called implicitly. This means that the following constructor declarations are effectively the same:

```
public Manager(int initialData)
{
    //Add further instructions here.
}
```

```
public Manager(int initialData)
    : base()
{
    //Add further instructions here.
}
```

If a base class does not offer a parameterless constructor, the derived class must make an explicit call to a base constructor by using `base`.

A constructor can invoke another constructor in the same object by using the `this` keyword. Like `base`, `this` can be used with or without parameters, and any parameters in the constructor are available as parameters to `this`, or as part of an expression. For example, the second constructor in the previous example can be rewritten using `this`:

```
public Employee(int weeklySalary, int numberOfWeeks)
    : this(weeklySalary * numberOfWeeks)
{
}
```

The use of the `this` keyword in the previous example causes this constructor to be called:

```
public Employee(int annualSalary)
{
    Salary = annualSalary;
}
```

Constructors can be marked as [public](#), [private](#), [protected](#), [internal](#), [protected internal](#) or [private protected](#). These access modifiers define how users of the class can construct the class. For more information, see [Access Modifiers](#).

A constructor can be declared static by using the [static](#) keyword. Static constructors are called automatically, immediately before any static fields are accessed, and are generally used to initialize static class members. For more information, see [Static Constructors](#).

C# Language Specification

For more information, see [Instance constructors](#) and [Static constructors](#) in the [C# Language Specification](#). The language specification is the definitive source for C# syntax and usage.

See also

- [C# Programming Guide](#)
- [The C# type system](#)
- [Constructors](#)
- [Finalizers](#)

Instance constructors (C# programming guide)

12/28/2021 • 2 minutes to read • [Edit Online](#)

You declare an instance constructor to specify the code that is executed when you create a new instance of a type with the `new` expression. To initialize a `static` class or static variables in a non-static class, you can define a `static constructor`.

As the following example shows, you can declare several instance constructors in one type:

```
using System;

class Coords
{
    public Coords()
        : this(0, 0)
    { }

    public Coords(int x, int y)
    {
        X = x;
        Y = y;
    }

    public int X { get; set; }
    public int Y { get; set; }

    public override string ToString() => $"({X},{Y})";
}

class Example
{
    static void Main()
    {
        var p1 = new Coords();
        Console.WriteLine($"Coords #1 at {p1}");
        // Output: Coords #1 at (0,0)

        var p2 = new Coords(5, 3);
        Console.WriteLine($"Coords #2 at {p2}");
        // Output: Coords #2 at (5,3)
    }
}
```

In the preceding example, the first, parameterless, constructor calls the second constructor with both arguments equal `0`. To do that, use the `this` keyword.

When you declare an instance constructor in a derived class, you can call a constructor of a base class. To do that, use the `base` keyword, as the following example shows:

```

using System;

abstract class Shape
{
    public const double pi = Math.PI;
    protected double x, y;

    public Shape(double x, double y)
    {
        this.x = x;
        this.y = y;
    }

    public abstract double Area();
}

class Circle : Shape
{
    public Circle(double radius)
        : base(radius, 0)
    { }

    public override double Area() => pi * x * x;
}

class Cylinder : Circle
{
    public Cylinder(double radius, double height)
        : base(radius)
    {
        y = height;
    }

    public override double Area() => (2 * base.Area()) + (2 * pi * x * y);
}

class Example
{
    static void Main()
    {
        double radius = 2.5;
        double height = 3.0;

        var ring = new Circle(radius);
        Console.WriteLine($"Area of the circle = {ring.Area():F2}");
        // Output: Area of the circle = 19.63

        var tube = new Cylinder(radius, height);
        Console.WriteLine($"Area of the cylinder = {tube.Area():F2}");
        // Output: Area of the cylinder = 86.39
    }
}

```

Parameterless constructors

If a *class* has no explicit instance constructors, C# provides a parameterless constructor that you can use to instantiate an instance of that class, as the following example shows:

```

using System;

public class Person
{
    public int age;
    public string name = "unknown";
}

class Example
{
    static void Main()
    {
        var person = new Person();
        Console.WriteLine($"Name: {person.name}, Age: {person.age}");
        // Output: Name: unknown, Age: 0
    }
}

```

That constructor initializes instance fields and properties according to the corresponding initializers. If a field or property has no initializer, its value is set to the [default value](#) of the field's or property's type. If you declare at least one instance constructor in a class, C# doesn't provide a parameterless constructor.

A *structure* type always provides a parameterless constructor as follows:

- In C# 9.0 and earlier, that is an implicit parameterless constructor that produces the [default value](#) of a type.
- In C# 10 and later, that is either an implicit parameterless constructor that produces the default value of a type or an explicitly declared parameterless constructor. For more information, see the [Parameterless constructors and field initializers](#) section of the [Structure types](#) article.

See also

- [C# programming guide](#)
- [Classes, structs, and records](#)
- [Constructors](#)
- [Finalizers](#)
- [base](#)
- [this](#)

Private Constructors (C# Programming Guide)

12/28/2021 • 2 minutes to read • [Edit Online](#)

A private constructor is a special instance constructor. It is generally used in classes that contain static members only. If a class has one or more private constructors and no public constructors, other classes (except nested classes) cannot create instances of this class. For example:

```
class NLog
{
    // Private Constructor:
    private NLog() { }

    public static double e = Math.E; //2.71828...
}
```

The declaration of the empty constructor prevents the automatic generation of a parameterless constructor. Note that if you do not use an access modifier with the constructor it will still be private by default. However, the [private](#) modifier is usually used explicitly to make it clear that the class cannot be instantiated.

Private constructors are used to prevent creating instances of a class when there are no instance fields or methods, such as the [Math](#) class, or when a method is called to obtain an instance of a class. If all the methods in the class are static, consider making the complete class static. For more information see [Static Classes and Static Class Members](#).

Example

The following is an example of a class using a private constructor.


```

public class Counter
{
    private Counter() { }

    public static int currentCount;

    public static int IncrementCount()
    {
        return ++currentCount;
    }
}

class TestCounter
{
    static void Main()
    {
        // If you uncomment the following statement, it will generate
        // an error because the constructor is inaccessible:
        // Counter aCounter = new Counter();    // Error

        Counter.currentCount = 100;
        Counter.IncrementCount();
        Console.WriteLine("New count: {0}", Counter.currentCount);

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
// Output: New count: 101

```

Notice that if you uncomment the following statement from the example, it will generate an error because the constructor is inaccessible because of its protection level:

```

// Counter aCounter = new Counter();    // Error

```

C# Language Specification

For more information, see [Private constructors](#) in the [C# Language Specification](#). The language specification is the definitive source for C# syntax and usage.

See also

- [C# Programming Guide](#)
- [The C# type system](#)
- [Constructors](#)
- [Finalizers](#)
- [private](#)
- [public](#)

Static Constructors (C# Programming Guide)

12/28/2021 • 4 minutes to read • [Edit Online](#)

A static constructor is used to initialize any [static](#) data, or to perform a particular action that needs to be performed only once. It is called automatically before the first instance is created or any static members are referenced.

```
class SimpleClass
{
    // Static variable that must be initialized at run time.
    static readonly long baseline;

    // Static constructor is called at most one time, before any
    // instance constructor is invoked or member is accessed.
    static SimpleClass()
    {
        baseline = DateTime.Now.Ticks;
    }
}
```

Remarks

Static constructors have the following properties:

- A static constructor doesn't take access modifiers or have parameters.
- A class or struct can only have one static constructor.
- Static constructors cannot be inherited or overloaded.
- A static constructor cannot be called directly and is only meant to be called by the common language runtime (CLR). It is invoked automatically.
- The user has no control on when the static constructor is executed in the program.
- A static constructor is called automatically. It initializes the [class](#) before the first instance is created or any static members declared in that class (not its base classes) are referenced. A static constructor runs before an instance constructor. A type's static constructor is called when a static method assigned to an event or a delegate is invoked and not when it is assigned. If static field variable initializers are present in the class of the static constructor, they're executed in the textual order in which they appear in the class declaration. The initializers run immediately prior to the execution of the static constructor.
- If you don't provide a static constructor to initialize static fields, all static fields are initialized to their default value as listed in [Default values of C# types](#).
- If a static constructor throws an exception, the runtime doesn't invoke it a second time, and the type will remain uninitialized for the lifetime of the application domain. Most commonly, a [TypeInitializationException](#) exception is thrown when a static constructor is unable to instantiate a type or for an unhandled exception occurring within a static constructor. For static constructors that aren't explicitly defined in source code, troubleshooting may require inspection of the intermediate language (IL) code.
- The presence of a static constructor prevents the addition of the [BeforeFieldInit](#) type attribute. This limits runtime optimization.
- A field declared as `static readonly` may only be assigned as part of its declaration or in a static constructor. When an explicit static constructor isn't required, initialize static fields at declaration rather than through a static constructor for better runtime optimization.
- The runtime calls a static constructor no more than once in a single application domain. That call is made in a locked region based on the specific type of the class. No additional locking mechanisms are needed in the

body of a static constructor. To avoid the risk of deadlocks, don't block the current thread in static constructors and initializers. For example, don't wait on tasks, threads, wait handles or events, don't acquire locks, and don't execute blocking parallel operations such as parallel loops, `Parallel.Invoke` and Parallel LINQ queries.

NOTE

Though not directly accessible, the presence of an explicit static constructor should be documented to assist with troubleshooting initialization exceptions.

Usage

- A typical use of static constructors is when the class is using a log file and the constructor is used to write entries to this file.
- Static constructors are also useful when creating wrapper classes for unmanaged code, when the constructor can call the `LoadLibrary` method.
- Static constructors are also a convenient place to enforce run-time checks on the type parameter that cannot be checked at compile time via type-parameter constraints.

Example

In this example, class `Bus` has a static constructor. When the first instance of `Bus` is created (`bus1`), the static constructor is invoked to initialize the class. The sample output verifies that the static constructor runs only one time, even though two instances of `Bus` are created, and that it runs before the instance constructor runs.

```
public class Bus
{
    // Static variable used by all Bus instances.
    // Represents the time the first bus of the day starts its route.
    protected static readonly DateTime globalStartTime;

    // Property for the number of each bus.
    protected int RouteNumber { get; set; }

    // Static constructor to initialize the static variable.
    // It is invoked before the first instance constructor is run.
    static Bus()
    {
        globalStartTime = DateTime.Now;

        // The following statement produces the first line of output,
        // and the line occurs only once.
        Console.WriteLine("Static constructor sets global start time to {0}",
            globalStartTime.ToLongTimeString());
    }

    // Instance constructor.
    public Bus(int routeNum)
    {
        RouteNumber = routeNum;
        Console.WriteLine("Bus #{0} is created.", RouteNumber);
    }

    // Instance method.
    public void Drive()
    {
        TimeSpan elapsedTime = DateTime.Now - globalStartTime;

        // For demonstration purposes we treat milliseconds as minutes to simulate
        // actual bus times. Do not do this in your actual bus schedule program!
        Console.WriteLine("{0} is starting its route {1:N2} minutes after global start time {2}.",
```

```

        this.RouteNumber,
        elapsedTime.Milliseconds,
        globalStartTime.ToShortTimeString());
    }
}

class TestBus
{
    static void Main()
    {
        // The creation of this instance activates the static constructor.
        Bus bus1 = new Bus(71);

        // Create a second bus.
        Bus bus2 = new Bus(72);

        // Send bus1 on its way.
        bus1.Drive();

        // Wait for bus2 to warm up.
        System.Threading.Thread.Sleep(25);

        // Send bus2 on its way.
        bus2.Drive();

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}

/* Sample output:
    Static constructor sets global start time to 3:57:08 PM.
    Bus #71 is created.
    Bus #72 is created.
    71 is starting its route 6.00 minutes after global start time 3:57 PM.
    72 is starting its route 31.00 minutes after global start time 3:57 PM.
*/

```

C# language specification

For more information, see the [Static constructors](#) section of the [C# language specification](#).

See also

- [C# Programming Guide](#)
- [The C# type system](#)
- [Constructors](#)
- [Static Classes and Static Class Members](#)
- [Finalizers](#)
- [Constructor Design Guidelines](#)
- [Security Warning - CA2121: Static constructors should be private](#)
- [Module initializers](#)

How to write a copy constructor (C# Programming Guide)

12/28/2021 • 2 minutes to read • [Edit Online](#)

C# [records](#) provide a copy constructor for objects, but for classes you have to write one yourself.

Example

In the following example, the `Person` class defines a copy constructor that takes, as its argument, an instance of `Person`. The values of the properties of the argument are assigned to the properties of the new instance of `Person`. The code contains an alternative copy constructor that sends the `Name` and `Age` properties of the instance that you want to copy to the instance constructor of the class.

```

using System;

class Person
{
    // Copy constructor.
    public Person(Person previousPerson)
    {
        Name = previousPerson.Name;
        Age = previousPerson.Age;
    }

    //// Alternate copy constructor calls the instance constructor.
    //public Person(Person previousPerson)
    //    : this(previousPerson.Name, previousPerson.Age)
    //{
    //}

    // Instance constructor.
    public Person(string name, int age)
    {
        Name = name;
        Age = age;
    }

    public int Age { get; set; }

    public string Name { get; set; }

    public string Details()
    {
        return Name + " is " + Age.ToString();
    }
}

class TestPerson
{
    static void Main()
    {
        // Create a Person object by using the instance constructor.
        Person person1 = new Person("George", 40);

        // Create another Person object, copying person1.
        Person person2 = new Person(person1);

        // Change each person's age.
        person1.Age = 39;
        person2.Age = 41;

        // Change person2's name.
        person2.Name = "Charles";

        // Show details to verify that the name and age fields are distinct.
        Console.WriteLine(person1.Details());
        Console.WriteLine(person2.Details());

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}

// Output:
// George is 39
// Charles is 41

```

See also

- [ICloneable](#)
- [Records](#)
- [C# Programming Guide](#)
- [The C# type system](#)
- [Constructors](#)
- [Finalizers](#)

Finalizers (C# Programming Guide)

12/28/2021 • 4 minutes to read • [Edit Online](#)

Finalizers (historically referred to as **destructors**) are used to perform any necessary final clean-up when a class instance is being collected by the garbage collector. In most cases, you can avoid writing a finalizer by using the [System.Runtime.InteropServices.SafeHandle](#) or derived classes to wrap any unmanaged handle.

Remarks

- Finalizers cannot be defined in structs. They are only used with classes.
- A class can only have one finalizer.
- Finalizers cannot be inherited or overloaded.
- Finalizers cannot be called. They are invoked automatically.
- A finalizer does not take modifiers or have parameters.

For example, the following is a declaration of a finalizer for the `Car` class.

```
class Car
{
    ~Car() // finalizer
    {
        // cleanup statements...
    }
}
```

A finalizer can also be implemented as an expression body definition, as the following example shows.

```
using System;

public class Destroyer
{
    public override string ToString() => GetType().Name;

    ~Destroyer() => Console.WriteLine($"The {ToString()} finalizer is executing.");
}
```

The finalizer implicitly calls [Finalize](#) on the base class of the object. Therefore, a call to a finalizer is implicitly translated to the following code:

```
protected override void Finalize()
{
    try
    {
        // Cleanup statements...
    }
    finally
    {
        base.Finalize();
    }
}
```

This design means that the `Finalize` method is called recursively for all instances in the inheritance chain, from the most-derived to the least-derived.

NOTE

Empty finalizers should not be used. When a class contains a finalizer, an entry is created in the `Finalize` queue. This queue is processed the garbage collector. When the GC processes the queue, it calls each finalizer. Unnecessary finalizers, including empty finalizers, finalizers that only call the base class finalizer, or finalizers that only call conditionally emitted methods cause a needless loss of performance.

The programmer has no control over when the finalizer is called; the garbage collector decides when to call it. The garbage collector checks for objects that are no longer being used by the application. If it considers an object eligible for finalization, it calls the finalizer (if any) and reclaims the memory used to store the object. It's possible to force garbage collection by calling `Collect`, but most of the time, this call should be avoided because it may create performance issues.

NOTE

Whether or not finalizers are run as part of application termination is specific to each [implementation of .NET](#). When an application terminates, .NET Framework makes every reasonable effort to call finalizers for objects that haven't yet been garbage collected, unless such cleanup has been suppressed (by a call to the library method `GC.SuppressFinalize`, for example). .NET 5 (including .NET Core) and later versions don't call finalizers as part of application termination. For more information, see GitHub issue [dotnet/csharpstandard #291](#).

If you need to perform cleanup reliably when an application exits, register a handler for the `System.AppDomain.ProcessExit` event. That handler would ensure `IDisposable.Dispose()` (or, `IAsyncDisposable.DisposeAsync()`) has been called for all objects that require cleanup before application exit. Because you can't call `Finalize` directly, and you can't guarantee the garbage collector calls all finalizers before exit, you must use `Dispose` or `DisposeAsync` to ensure resources are freed.

Using finalizers to release resources

In general, C# does not require as much memory management on the part of the developer as languages that don't target a runtime with garbage collection. This is because the .NET garbage collector implicitly manages the allocation and release of memory for your objects. However, when your application encapsulates unmanaged resources, such as windows, files, and network connections, you should use finalizers to free those resources. When the object is eligible for finalization, the garbage collector runs the `Finalize` method of the object.

Explicit release of resources

If your application is using an expensive external resource, we also recommend that you provide a way to explicitly release the resource before the garbage collector frees the object. To release the resource, implement a `Dispose` method from the `IDisposable` interface that performs the necessary cleanup for the object. This can considerably improve the performance of the application. Even with this explicit control over resources, the finalizer becomes a safeguard to clean up resources if the call to the `Dispose` method fails.

For more information about cleaning up resources, see the following articles:

- [Cleaning Up Unmanaged Resources](#)
- [Implementing a Dispose Method](#)
- [Implementing a DisposeAsync Method](#)
- [using Statement](#)

Example

The following example creates three classes that make a chain of inheritance. The class `First` is the base class, `Second` is derived from `First`, and `Third` is derived from `Second`. All three have finalizers. In `Main`, an instance of the most-derived class is created. The output from this code depends on which implementation of .NET the application targets:

- .NET Framework: The output shows that the finalizers for the three classes are called automatically when the application terminates, in order from the most-derived to the least-derived.
- .NET 5 (including .NET Core) or a later version: There's no output, because this implementation of .NET doesn't call finalizers when the application terminates.

```
class First
{
    ~First()
    {
        System.Diagnostics.Trace.WriteLine("First's finalizer is called.");
    }
}

class Second : First
{
    ~Second()
    {
        System.Diagnostics.Trace.WriteLine("Second's finalizer is called.");
    }
}

class Third : Second
{
    ~Third()
    {
        System.Diagnostics.Trace.WriteLine("Third's finalizer is called.");
    }
}

/*
Test with code like the following:
    Third t = new Third();
    t = null;

When objects are finalized, the output would be:
Third's finalizer is called.
Second's finalizer is called.
First's finalizer is called.
*/
```

C# language specification

For more information, see the [Destructors](#) section of the [C# language specification](#).

See also

- [IDisposable](#)
- [C# Programming Guide](#)
- [Constructors](#)
- [Garbage Collection](#)

Object and Collection Initializers (C# Programming Guide)

12/28/2021 • 9 minutes to read • [Edit Online](#)

C# lets you instantiate an object or collection and perform member assignments in a single statement.

Object initializers

Object initializers let you assign values to any accessible fields or properties of an object at creation time without having to invoke a constructor followed by lines of assignment statements. The object initializer syntax enables you to specify arguments for a constructor or omit the arguments (and parentheses syntax). The following example shows how to use an object initializer with a named type, `Cat` and how to invoke the parameterless constructor. Note the use of auto-implemented properties in the `Cat` class. For more information, see [Auto-Implemented Properties](#).

```
public class Cat
{
    // Auto-implemented properties.
    public int Age { get; set; }
    public string Name { get; set; }

    public Cat()
    {
    }

    public Cat(string name)
    {
        this.Name = name;
    }
}
```

```
Cat cat = new Cat { Age = 10, Name = "Fluffy" };
Cat sameCat = new Cat("Fluffy"){ Age = 10 };
```

The object initializers syntax allows you to create an instance, and after that it assigns the newly created object, with its assigned properties, to the variable in the assignment.

Starting with C# 6, object initializers can set indexers, in addition to assigning fields and properties. Consider this basic `Matrix` class:

```
public class Matrix
{
    private double[,] storage = new double[3, 3];

    public double this[int row, int column]
    {
        // The embedded array will throw out of range exceptions as appropriate.
        get { return storage[row, column]; }
        set { storage[row, column] = value; }
    }
}
```

You could initialize the identity matrix with the following code:

```
var identity = new Matrix
{
    [0, 0] = 1.0,
    [0, 1] = 0.0,
    [0, 2] = 0.0,

    [1, 0] = 0.0,
    [1, 1] = 1.0,
    [1, 2] = 0.0,

    [2, 0] = 0.0,
    [2, 1] = 0.0,
    [2, 2] = 1.0,
};
```

Any accessible indexer that contains an accessible setter can be used as one of the expressions in an object initializer, regardless of the number or types of arguments. The index arguments form the left side of the assignment, and the value is the right side of the expression. For example, these are all valid if `IndexersExample` has the appropriate indexers:

```
var thing = new IndexersExample {
    name = "object one",
    [1] = '1',
    [2] = '4',
    [3] = '9',
    Size = Math.PI,
    ['C',4] = "Middle C"
}
```

For the preceding code to compile, the `IndexersExample` type must have the following members:

```
public string name;
public double Size { set { ... }; }
public char this[int i] { set { ... }; }
public string this[char c, int i] { set { ... }; }
```

Object Initializers with anonymous types

Although object initializers can be used in any context, they are especially useful in LINQ query expressions. Query expressions make frequent use of [anonymous types](#), which can only be initialized by using an object initializer, as shown in the following declaration.

```
var pet = new { Age = 10, Name = "Fluffy" };
```

Anonymous types enable the `select` clause in a LINQ query expression to transform objects of the original sequence into objects whose value and shape may differ from the original. This is useful if you want to store only a part of the information from each object in a sequence. In the following example, assume that a product object (`p`) contains many fields and methods, and that you are only interested in creating a sequence of objects that contain the product name and the unit price.

```
var productInfos =
    from p in products
    select new { p.ProductName, p.UnitPrice };
```

When this query is executed, the `productInfos` variable will contain a sequence of objects that can be accessed in a `foreach` statement as shown in this example:

```
foreach(var p in productInfos){...}
```

Each object in the new anonymous type has two public properties that receive the same names as the properties or fields in the original object. You can also rename a field when you are creating an anonymous type; the following example renames the `UnitPrice` field to `Price`.

```
select new {p.ProductName, Price = p.UnitPrice};
```

Collection initializers

Collection initializers let you specify one or more element initializers when you initialize a collection type that implements [IEnumerable](#) and has `Add` with the appropriate signature as an instance method or an extension method. The element initializers can be a simple value, an expression, or an object initializer. By using a collection initializer, you do not have to specify multiple calls; the compiler adds the calls automatically.

The following example shows two simple collection initializers:

```
List<int> digits = new List<int> { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
List<int> digits2 = new List<int> { 0 + 1, 12 % 3, MakeInt() };
```

The following collection initializer uses object initializers to initialize objects of the `Cat` class defined in a previous example. Note that the individual object initializers are enclosed in braces and separated by commas.

```
List<Cat> cats = new List<Cat>
{
    new Cat{ Name = "Sylvester", Age=8 },
    new Cat{ Name = "Whiskers", Age=2 },
    new Cat{ Name = "Sasha", Age=14 }
};
```

You can specify `null` as an element in a collection initializer if the collection's `Add` method allows it.

```
List<Cat> moreCats = new List<Cat>
{
    new Cat{ Name = "Furrytail", Age=5 },
    new Cat{ Name = "Peaches", Age=4 },
    null
};
```

You can specify indexed elements if the collection supports read / write indexing.

```
var numbers = new Dictionary<int, string>
{
    [7] = "seven",
    [9] = "nine",
    [13] = "thirteen"
};
```

The preceding sample generates code that calls the `Item[TKey]` to set the values. Before C# 6, you could initialize dictionaries and other associative containers using the following syntax. Notice that instead of indexer syntax,

with parentheses and an assignment, it uses an object with multiple values:

```
var moreNumbers = new Dictionary<int, string>
{
    {19, "nineteen" },
    {23, "twenty-three" },
    {42, "forty-two" }
};
```

This initializer example calls [Add\(TKey, TValue\)](#) to add the three items into the dictionary. These two different ways to initialize associative collections have slightly different behavior because of the method calls the compiler generates. Both variants work with the `Dictionary` class. Other types may only support one or the other based on their public API.

Object Initializers with collection read-only property initialization

Some classes may have collection properties where the property is read-only, like the `Cats` property of `CatOwner` in the following case:

```
public class CatOwner
{
    public IList<Cat> Cats { get; } = new List<Cat>();
}
```

You will not be able to use collection initializer syntax discussed so far since the property cannot be assigned a new list:

```
CatOwner owner = new CatOwner
{
    Cats = new List<Cat>
    {
        new Cat{ Name = "Sylvester", Age=8 },
        new Cat{ Name = "Whiskers", Age=2 },
        new Cat{ Name = "Sasha", Age=14 }
    }
};
```

However, new entries can be added to `Cats` nonetheless using the initialization syntax by omitting the list creation (`new List<Cat>`), as shown next:

```
CatOwner owner = new CatOwner
{
    Cats =
    {
        new Cat{ Name = "Sylvester", Age=8 },
        new Cat{ Name = "Whiskers", Age=2 },
        new Cat{ Name = "Sasha", Age=14 }
    }
};
```

The set of entries to be added simply appear surrounded by braces. The above is identical to writing:

```
CatOwner owner = new CatOwner();
owner.Cats.Add(new Cat{ Name = "Sylvester", Age=8 });
owner.Cats.Add(new Cat{ Name = "Whiskers", Age=2 });
owner.Cats.Add(new Cat{ Name = "Sasha", Age=14 });
```

Examples

The following example combines the concepts of object and collection initializers.

```
public class InitializationSample
{
    public class Cat
    {
        // Auto-implemented properties.
        public int Age { get; set; }
        public string Name { get; set; }

        public Cat() { }

        public Cat(string name)
        {
            Name = name;
        }
    }

    public static void Main()
    {
        Cat cat = new Cat { Age = 10, Name = "Fluffy" };
        Cat sameCat = new Cat("Fluffy"){ Age = 10 };

        List<Cat> cats = new List<Cat>
        {
            new Cat { Name = "Sylvester", Age = 8 },
            new Cat { Name = "Whiskers", Age = 2 },
            new Cat { Name = "Sasha", Age = 14 }
        };

        List<Cat> moreCats = new List<Cat>
        {
            new Cat { Name = "Furrytail", Age = 5 },
            new Cat { Name = "Peaches", Age = 4 },
            null
        };

        // Display results.
        System.Console.WriteLine(cat.Name);

        foreach (Cat c in cats)
            System.Console.WriteLine(c.Name);

        foreach (Cat c in moreCats)
            if (c != null)
                System.Console.WriteLine(c.Name);
            else
                System.Console.WriteLine("List element has null value.");
    }
    // Output:
    //Fluffy
    //Sylvester
    //Whiskers
    //Sasha
    //Furrytail
    //Peaches
    //List element has null value.
}
```

The following example shows an object that implements [IEnumerable](#) and contains an `Add` method with multiple parameters. It uses a collection initializer with multiple elements per item in the list that correspond to the signature of the `Add` method.

```

public class FullExample
{
    class FormattedAddresses : IEnumerable<string>
    {
        private List<string> internalList = new List<string>();
        public IEnumerator<string> GetEnumerator() => internalList.GetEnumerator();

        System.Collections.IEnumerator System.Collections.IEnumerable.GetEnumerator() =>
internalList.GetEnumerator();

        public void Add(string firstname, string lastname,
            string street, string city,
            string state, string zipcode) => internalList.Add(
                $"{firstname} {lastname}
{street}
{city}, {state} {zipcode}"
            );
    }

    public static void Main()
    {
        FormattedAddresses addresses = new FormattedAddresses()
        {
            {"John", "Doe", "123 Street", "Topeka", "KS", "00000" },
            {"Jane", "Smith", "456 Street", "Topeka", "KS", "00000" }
        };

        Console.WriteLine("Address Entries:");

        foreach (string addressEntry in addresses)
        {
            Console.WriteLine("\r\n" + addressEntry);
        }
    }

    /*
    * Prints:

    Address Entries:

    John Doe
    123 Street
    Topeka, KS 00000

    Jane Smith
    456 Street
    Topeka, KS 00000
    */
}

```

`Add` methods can use the `params` keyword to take a variable number of arguments, as shown in the following example. This example also demonstrates the custom implementation of an indexer to initialize a collection using indexes.

```

public class DictionaryExample
{
    class RudimentaryMultiValuedDictionary<TKey, TValue> : IEnumerable<KeyValuePair<TKey, List<TValue>>>
    {
        private Dictionary<TKey, List<TValue>> internalDictionary = new Dictionary<TKey, List<TValue>>>();

        public IEnumerator<KeyValuePair<TKey, List<TValue>>> GetEnumerator() =>
internalDictionary.GetEnumerator();

        System.Collections.IEnumerator System.Collections.IEnumerable.GetEnumerator() =>
internalDictionary.GetEnumerator();
    }
}

```



```

public List<TValue> this[TKey key]
{
    get => internalDictionary[key];
    set => Add(key, value);
}

public void Add(TKey key, params TValue[] values) => Add(key, (IEnumerable<TValue>)values);

public void Add(TKey key, IEnumerable<TValue> values)
{
    if (!internalDictionary.TryGetValue(key, out List<TValue> storedValues))
        internalDictionary.Add(key, storedValues = new List<TValue>());

    storedValues.AddRange(values);
}
}

public static void Main()
{
    RudimentaryMultiValuedDictionary<string, string> rudimentaryMultiValuedDictionary1
        = new RudimentaryMultiValuedDictionary<string, string>()
        {
            {"Group1", "Bob", "John", "Mary" },
            {"Group2", "Eric", "Emily", "Debbie", "Jesse" }
        };

    RudimentaryMultiValuedDictionary<string, string> rudimentaryMultiValuedDictionary2
        = new RudimentaryMultiValuedDictionary<string, string>()
        {
            ["Group1"] = new List<string>() { "Bob", "John", "Mary" },
            ["Group2"] = new List<string>() { "Eric", "Emily", "Debbie", "Jesse" }
        };

    RudimentaryMultiValuedDictionary<string, string> rudimentaryMultiValuedDictionary3
        = new RudimentaryMultiValuedDictionary<string, string>()
        {
            {"Group1", new string []{ "Bob", "John", "Mary" } },
            {"Group2", new string []{ "Eric", "Emily", "Debbie", "Jesse" } }
        };

    Console.WriteLine("Using first multi-valued dictionary created with a collection initializer:");

    foreach (KeyValuePair<string, List<string>> group in rudimentaryMultiValuedDictionary1)
    {
        Console.WriteLine($"\"\\r\\nMembers of group {group.Key}: ");

        foreach (string member in group.Value)
        {
            Console.WriteLine(member);
        }
    }

    Console.WriteLine("\\r\\nUsing second multi-valued dictionary created with a collection initializer using indexing:");

    foreach (KeyValuePair<string, List<string>> group in rudimentaryMultiValuedDictionary2)
    {
        Console.WriteLine($"\"\\r\\nMembers of group {group.Key}: ");

        foreach (string member in group.Value)
        {
            Console.WriteLine(member);
        }
    }

    Console.WriteLine("\\r\\nUsing third multi-valued dictionary created with a collection initializer using indexing:");

    foreach (KeyValuePair<string, List<string>> group in rudimentaryMultiValuedDictionary3)
    {
        Console.WriteLine($"\"\\r\\nMembers of group {group.Key}: ");
    }
}

```

```

        foreach (string member in group.Value)
        {
            Console.WriteLine(member);
        }
    }
}

/*
 * Prints:

    Using first multi-valued dictionary created with a collection initializer:

    Members of group Group1:
    Bob
    John
    Mary

    Members of group Group2:
    Eric
    Emily
    Debbie
    Jesse

    Using second multi-valued dictionary created with a collection initializer using indexing:

    Members of group Group1:
    Bob
    John
    Mary

    Members of group Group2:
    Eric
    Emily
    Debbie
    Jesse

    Using third multi-valued dictionary created with a collection initializer using indexing:

    Members of group Group1:
    Bob
    John
    Mary

    Members of group Group2:
    Eric
    Emily
    Debbie
    Jesse
 */
}

```

See also

- [C# Programming Guide](#)
- [LINQ in C#](#)
- [Anonymous Types](#)

How to initialize objects by using an object initializer (C# Programming Guide)

12/28/2021 • 2 minutes to read • [Edit Online](#)

You can use object initializers to initialize type objects in a declarative manner without explicitly invoking a constructor for the type.

The following examples show how to use object initializers with named objects. The compiler processes object initializers by first accessing the parameterless instance constructor and then processing the member initializations. Therefore, if the parameterless constructor is declared as `private` in the class, object initializers that require public access will fail.

You must use an object initializer if you're defining an anonymous type. For more information, see [How to return subsets of element properties in a query](#).

Example

The following example shows how to initialize a new `StudentName` type by using object initializers. This example sets properties in the `StudentName` type:

```
public class HowToObjectInitializers
{
    public static void Main()
    {
        // Declare a StudentName by using the constructor that has two parameters.
        StudentName student1 = new StudentName("Craig", "Playstead");

        // Make the same declaration by using an object initializer and sending
        // arguments for the first and last names. The parameterless constructor is
        // invoked in processing this declaration, not the constructor that has
        // two parameters.
        StudentName student2 = new StudentName
        {
            FirstName = "Craig",
            LastName = "Playstead"
        };

        // Declare a StudentName by using an object initializer and sending
        // an argument for only the ID property. No corresponding constructor is
        // necessary. Only the parameterless constructor is used to process object
        // initializers.
        StudentName student3 = new StudentName
        {
            ID = 183
        };

        // Declare a StudentName by using an object initializer and sending
        // arguments for all three properties. No corresponding constructor is
        // defined in the class.
        StudentName student4 = new StudentName
        {
            FirstName = "Craig",
            LastName = "Playstead",
            ID = 116
        };

        Console.WriteLine(student1.ToString());
        Console.WriteLine(student2.ToString());
    }
}
```

```

        Console.WriteLine(student2.ToString());
        Console.WriteLine(student3.ToString());
        Console.WriteLine(student4.ToString());
    }
    // Output:
    // Craig 0
    // Craig 0
    // 183
    // Craig 116

    public class StudentName
    {
        // This constructor has no parameters. The parameterless constructor
        // is invoked in the processing of object initializers.
        // You can test this by changing the access modifier from public to
        // private. The declarations in Main that use object initializers will
        // fail.
        public StudentName() { }

        // The following constructor has parameters for two of the three
        // properties.
        public StudentName(string first, string last)
        {
            FirstName = first;
            LastName = last;
        }

        // Properties.
        public string FirstName { get; set; }
        public string LastName { get; set; }
        public int ID { get; set; }

        public override string ToString() => FirstName + " " + ID;
    }
}

```

Object initializers can be used to set indexers in an object. The following example defines a `BaseballTeam` class that uses an indexer to get and set players at different positions. The initializer can assign players, based on the abbreviation for the position, or the number used for each position baseball scorecards:

```

public class HowToIndexInitializer
{
    public class BaseballTeam
    {
        private string[] players = new string[9];
        private readonly List<string> positionAbbreviations = new List<string>
        {
            "P", "C", "1B", "2B", "3B", "SS", "LF", "CF", "RF"
        };

        public string this[int position]
        {
            // Baseball positions are 1 - 9.
            get { return players[position-1]; }
            set { players[position-1] = value; }
        }
        public string this[string position]
        {
            get { return players[positionAbbreviations.IndexOf(position)]; }
            set { players[positionAbbreviations.IndexOf(position)] = value; }
        }
    }

    public static void Main()
    {
        var team = new BaseballTeam
        {
            ["RF"] = "Mookie Betts",
            [4] = "Jose Altuve",
            ["CF"] = "Mike Trout"
        };

        Console.WriteLine(team["2B"]);
    }
}

```

See also

- [C# Programming Guide](#)
- [Object and Collection Initializers](#)

How to initialize a dictionary with a collection initializer (C# Programming Guide)

12/28/2021 • 2 minutes to read • [Edit Online](#)

A `Dictionary<TKey,TValue>` contains a collection of key/value pairs. Its `Add` method takes two parameters, one for the key and one for the value. One way to initialize a `Dictionary<TKey,TValue>`, or any collection whose `Add` method takes multiple parameters, is to enclose each set of parameters in braces as shown in the following example. Another option is to use an index initializer, also shown in the following example.

Example

In the following code example, a `Dictionary<TKey,TValue>` is initialized with instances of type `StudentName`. The first initialization uses the `Add` method with two arguments. The compiler generates a call to `Add` for each of the pairs of `int` keys and `StudentName` values. The second uses a public read / write indexer method of the `Dictionary` class:

```
public class HowToDictionaryInitializer
{
    class StudentName
    {
        public string FirstName { get; set; }
        public string LastName { get; set; }
        public int ID { get; set; }
    }

    public static void Main()
    {
        var students = new Dictionary<int, StudentName>()
        {
            { 111, new StudentName { FirstName="Sachin", LastName="Karnik", ID=211 } },
            { 112, new StudentName { FirstName="Dina", LastName="Salimzianova", ID=317 } },
            { 113, new StudentName { FirstName="Andy", LastName="Ruth", ID=198 } }
        };

        foreach(var index in Enumerable.Range(111, 3))
        {
            Console.WriteLine($"Student {index} is {students[index].FirstName} {students[index].LastName}");
        }
        Console.WriteLine();

        var students2 = new Dictionary<int, StudentName>()
        {
            [111] = new StudentName { FirstName="Sachin", LastName="Karnik", ID=211 },
            [112] = new StudentName { FirstName="Dina", LastName="Salimzianova", ID=317 } ,
            [113] = new StudentName { FirstName="Andy", LastName="Ruth", ID=198 }
        };

        foreach (var index in Enumerable.Range(111, 3))
        {
            Console.WriteLine($"Student {index} is {students2[index].FirstName} {students2[index].LastName}");
        }
    }
}
```

Note the two pairs of braces in each element of the collection in the first declaration. The innermost braces

enclose the object initializer for the `StudentName`, and the outermost braces enclose the initializer for the key/value pair that will be added to the `students` [Dictionary<TKey,TValue>](#). Finally, the whole collection initializer for the dictionary is enclosed in braces. In the second initialization, the left side of the assignment is the key and the right side is the value, using an object initializer for `StudentName`.

See also

- [C# Programming Guide](#)
- [Object and Collection Initializers](#)

Nested Types (C# Programming Guide)

12/28/2021 • 2 minutes to read • [Edit Online](#)

A type defined within a [class](#), [struct](#), or [interface](#) is called a nested type. For example

```
public class Container
{
    class Nested
    {
        Nested() { }
    }
}
```

Regardless of whether the outer type is a class, interface, or struct, nested types default to [private](#); they are accessible only from their containing type. In the previous example, the `Nested` class is inaccessible to external types.

You can also specify an [access modifier](#) to define the accessibility of a nested type, as follows:

- Nested types of a **class** can be [public](#), [protected](#), [internal](#), [protected internal](#), [private](#) or [private protected](#).

However, defining a `protected`, `protected internal` or `private protected` nested class inside a [sealed class](#) generates compiler warning [CS0628](#), "new protected member declared in sealed class."

Also be aware that making a nested type externally visible violates the code quality rule [CA1034](#) "Nested types should not be visible".

- Nested types of a **struct** can be [public](#), [internal](#), or [private](#).

The following example makes the `Nested` class public:

```
public class Container
{
    public class Nested
    {
        Nested() { }
    }
}
```

The nested, or inner, type can access the containing, or outer, type. To access the containing type, pass it as an argument to the constructor of the nested type. For example:


```
public class Container
{
    public class Nested
    {
        private Container parent;

        public Nested()
        {
        }
        public Nested(Container parent)
        {
            this.parent = parent;
        }
    }
}
```

A nested type has access to all of the members that are accessible to its containing type. It can access private and protected members of the containing type, including any inherited protected members.

In the previous declaration, the full name of class `Nested` is `Container.Nested`. This is the name used to create a new instance of the nested class, as follows:

```
Container.Nested nest = new Container.Nested();
```

See also

- [C# Programming Guide](#)
- [The C# type system](#)
- [Access Modifiers](#)
- [Constructors](#)
- [CA1034 rule](#)

Partial Classes and Methods (C# Programming Guide)

12/28/2021 • 6 minutes to read • [Edit Online](#)

It is possible to split the definition of a [class](#), a [struct](#), an [interface](#) or a method over two or more source files. Each source file contains a section of the type or method definition, and all parts are combined when the application is compiled.

Partial Classes

There are several situations when splitting a class definition is desirable:

- When working on large projects, spreading a class over separate files enables multiple programmers to work on it at the same time.
- When working with automatically generated source, code can be added to the class without having to recreate the source file. Visual Studio uses this approach when it creates Windows Forms, Web service wrapper code, and so on. You can create code that uses these classes without having to modify the file created by Visual Studio.
- When using [source generators](#) to generate additional functionality in a class.

To split a class definition, use the [partial](#) keyword modifier, as shown here:

```
public partial class Employee
{
    public void DoWork()
    {
    }
}

public partial class Employee
{
    public void GoToLunch()
    {
    }
}
```

The `partial` keyword indicates that other parts of the class, struct, or interface can be defined in the namespace. All the parts must use the `partial` keyword. All the parts must be available at compile time to form the final type. All the parts must have the same accessibility, such as `public`, `private`, and so on.

If any part is declared abstract, then the whole type is considered abstract. If any part is declared sealed, then the whole type is considered sealed. If any part declares a base type, then the whole type inherits that class.

All the parts that specify a base class must agree, but parts that omit a base class still inherit the base type. Parts can specify different base interfaces, and the final type implements all the interfaces listed by all the partial declarations. Any class, struct, or interface members declared in a partial definition are available to all the other parts. The final type is the combination of all the parts at compile time.

NOTE

The `partial` modifier is not available on delegate or enumeration declarations.

The following example shows that nested types can be partial, even if the type they are nested within is not partial itself.

```
class Container
{
    partial class Nested
    {
        void Test() { }
    }

    partial class Nested
    {
        void Test2() { }
    }
}
```

At compile time, attributes of partial-type definitions are merged. For example, consider the following declarations:

```
[SerializableAttribute]
partial class Moon { }

[ObsoleteAttribute]
partial class Moon { }
```

They are equivalent to the following declarations:

```
[SerializableAttribute]
[ObsoleteAttribute]
class Moon { }
```

The following are merged from all the partial-type definitions:

- XML comments
- interfaces
- generic-type parameter attributes
- class attributes
- members

For example, consider the following declarations:

```
partial class Earth : Planet, IRotate { }
partial class Earth : IRevolve { }
```

They are equivalent to the following declarations:

```
class Earth : Planet, IRotate, IRevolve { }
```

Restrictions

There are several rules to follow when you are working with partial class definitions:

- All partial-type definitions meant to be parts of the same type must be modified with `partial`. For example, the following class declarations generate an error:

```
public partial class A { }  
//public class A { } // Error, must also be marked partial
```

- The `partial` modifier can only appear immediately before the keywords `class`, `struct`, or `interface`.
- Nested partial types are allowed in partial-type definitions as illustrated in the following example:

```
partial class ClassWithNestedClass  
{  
    partial class NestedClass { }  
}  
  
partial class ClassWithNestedClass  
{  
    partial class NestedClass { }  
}
```

- All partial-type definitions meant to be parts of the same type must be defined in the same assembly and the same module (.exe or .dll file). Partial definitions cannot span multiple modules.
- The class name and generic-type parameters must match on all partial-type definitions. Generic types can be partial. Each partial declaration must use the same parameter names in the same order.
- The following keywords on a partial-type definition are optional, but if present on one partial-type definition, cannot conflict with the keywords specified on another partial definition for the same type:
 - `public`
 - `private`
 - `protected`
 - `internal`
 - `abstract`
 - `sealed`
 - base class
 - `new` modifier (nested parts)
 - generic constraints

For more information, see [Constraints on Type Parameters](#).

Examples

In the following example, the fields and the constructor of the class, `Coords`, are declared in one partial class definition, and the member, `PrintCoords`, is declared in another partial class definition.

```

public partial class Coords
{
    private int x;
    private int y;

    public Coords(int x, int y)
    {
        this.x = x;
        this.y = y;
    }
}

public partial class Coords
{
    public void PrintCoords()
    {
        Console.WriteLine("Coords: {0},{1}", x, y);
    }
}

class TestCoords
{
    static void Main()
    {
        Coords myCoords = new Coords(10, 15);
        myCoords.PrintCoords();

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
// Output: Coords: 10,15

```

The following example shows that you can also develop partial structs and interfaces.

```

partial interface ITest
{
    void Interface_Test();
}

partial interface ITest
{
    void Interface_Test2();
}

partial struct S1
{
    void Struct_Test() { }
}

partial struct S1
{
    void Struct_Test2() { }
}

```

Partial Methods

A partial class or struct may contain a partial method. One part of the class contains the signature of the method. An implementation can be defined in the same part or another part. If the implementation is not supplied, then the method and all calls to the method are removed at compile time. Implementation may be required depending on method signature. A partial method isn't required to have an implementation in the

following cases:

- It doesn't have any accessibility modifiers (including the default [private](#)).
- It returns [void](#).
- It doesn't have any [out](#) parameters.
- It doesn't have any of the following modifiers [virtual](#), [override](#), [sealed](#), [new](#), or [extern](#).

Any method that doesn't conform to all those restrictions (for example, `public virtual partial void method`), must provide an implementation. That implementation may be supplied by a *source generator*.

Partial methods enable the implementer of one part of a class to declare a method. The implementer of another part of the class can define that method. There are two scenarios where this is useful: templates that generate boilerplate code, and source generators.

- **Template code:** The template reserves a method name and signature so that generated code can call the method. These methods follow the restrictions that enable a developer to decide whether to implement the method. If the method is not implemented, then the compiler removes the method signature and all calls to the method. The calls to the method, including any results that would occur from evaluation of arguments in the calls, have no effect at run time. Therefore, any code in the partial class can freely use a partial method, even if the implementation is not supplied. No compile-time or run-time errors will result if the method is called but not implemented.
- **Source generators:** Source generators provide an implementation for methods. The human developer can add the method declaration (often with attributes read by the source generator). The developer can write code that calls these methods. The source generator runs during compilation and provides the implementation. In this scenario, the restrictions for partial methods that may not be implemented often aren't followed.

```
// Definition in file1.cs
partial void OnNameChanged();

// Implementation in file2.cs
partial void OnNameChanged()
{
    // method body
}
```

- Partial method declarations must begin with the contextual keyword [partial](#).
- Partial method signatures in both parts of the partial type must match.
- Partial methods can have [static](#) and [unsafe](#) modifiers.
- Partial methods can be generic. Constraints are put on the defining partial method declaration, and may optionally be repeated on the implementing one. Parameter and type parameter names do not have to be the same in the implementing declaration as in the defining one.
- You can make a [delegate](#) to a partial method that has been defined and implemented, but not to a partial method that has only been defined.

C# Language Specification

For more information, see [Partial types](#) in the [C# Language Specification](#). The language specification is the definitive source for C# syntax and usage.

See also

- [C# Programming Guide](#)
- [Classes](#)

- [Structure types](#)
- [Interfaces](#)
- [partial \(Type\)](#)

How to return subsets of element properties in a query (C# Programming Guide)

12/28/2021 • 2 minutes to read • [Edit Online](#)

Use an anonymous type in a query expression when both of these conditions apply:

- You want to return only some of the properties of each source element.
- You do not have to store the query results outside the scope of the method in which the query is executed.

If you only want to return one property or field from each source element, then you can just use the dot operator in the `select` clause. For example, to return only the `ID` of each `student`, write the `select` clause as follows:

```
select student.ID;
```

Example

The following example shows how to use an anonymous type to return only a subset of the properties of each source element that matches the specified condition.

```
private static void QueryByScore()
{
    // Create the query. var is required because
    // the query produces a sequence of anonymous types.
    var queryHighScores =
        from student in students
        where student.ExamScores[0] > 95
        select new { student.FirstName, student.LastName };

    // Execute the query.
    foreach (var obj in queryHighScores)
    {
        // The anonymous type's properties were not named. Therefore
        // they have the same names as the Student properties.
        Console.WriteLine(obj.FirstName + ", " + obj.LastName);
    }
}

/* Output:
Adams, Terry
Fakhouri, Fadi
Garcia, Cesar
Omelchenko, Svetlana
Zabokritski, Eugene
*/
```

Note that the anonymous type uses the source element's names for its properties if no names are specified. To give new names to the properties in the anonymous type, write the `select` statement as follows:

```
select new { First = student.FirstName, Last = student.LastName };
```

If you try this in the previous example, then the `Console.WriteLine` statement must also change:


```
Console.WriteLine(student.First + " " + student.Last);
```

Compiling the Code

To run this code, copy and paste the class into a C# console application with a `using` directive for `System.Linq`.

See also

- [C# Programming Guide](#)
- [Anonymous Types](#)
- [LINQ in C#](#)

Explicit Interface Implementation (C# Programming Guide)

12/28/2021 • 2 minutes to read • [Edit Online](#)

If a [class](#) implements two interfaces that contain a member with the same signature, then implementing that member on the class will cause both interfaces to use that member as their implementation. In the following example, all the calls to `Paint` invoke the same method. This first sample defines the types:

```
public interface IControl
{
    void Paint();
}
public interface ISurface
{
    void Paint();
}
public class SampleClass : IControl, ISurface
{
    // Both ISurface.Paint and IControl.Paint call this method.
    public void Paint()
    {
        Console.WriteLine("Paint method in SampleClass");
    }
}
```

The following sample calls the methods:

```
SampleClass sample = new SampleClass();
IControl control = sample;
ISurface surface = sample;

// The following lines all call the same method.
sample.Paint();
control.Paint();
surface.Paint();

// Output:
// Paint method in SampleClass
// Paint method in SampleClass
// Paint method in SampleClass
```

But you might not want the same implementation to be called for both interfaces. To call a different implementation depending on which interface is in use, you can implement an interface member explicitly. An explicit interface implementation is a class member that is only called through the specified interface. Name the class member by prefixing it with the name of the interface and a period. For example:

```
public class SampleClass : IControl, ISurface
{
    void IControl.Paint()
    {
        System.Console.WriteLine("IControl.Paint");
    }
    void ISurface.Paint()
    {
        System.Console.WriteLine("ISurface.Paint");
    }
}
```

The class member `IControl.Paint` is only available through the `IControl` interface, and `ISurface.Paint` is only available through `ISurface`. Both method implementations are separate, and neither are available directly on the class. For example:

```
SampleClass sample = new SampleClass();
IControl control = sample;
ISurface surface = sample;

// The following lines all call the same method.
//sample.Paint(); // Compiler error.
control.Paint(); // Calls IControl.Paint on SampleClass.
surface.Paint(); // Calls ISurface.Paint on SampleClass.

// Output:
// IControl.Paint
// ISurface.Paint
```

Explicit implementation is also used to resolve cases where two interfaces each declare different members of the same name such as a property and a method. To implement both interfaces, a class has to use explicit implementation either for the property `P`, or the method `P`, or both, to avoid a compiler error. For example:

```
interface ILeft
{
    int P { get; }
}
interface IRight
{
    int P();
}

class Middle : ILeft, IRight
{
    public int P() { return 0; }
    int ILeft.P { get { return 0; } }
}
```

An explicit interface implementation doesn't have an access modifier since it isn't accessible as a member of the type it's defined in. Instead, it's only accessible when called through an instance of the interface. If you specify an access modifier for an explicit interface implementation, you get compiler error [CS0106](#). For more information, see [interface](#) (C# Reference).

Beginning with [C# 8.0](#), you can define an implementation for members declared in an interface. If a class inherits a method implementation from an interface, that method is only accessible through a reference of the interface type. The inherited member doesn't appear as part of the public interface. The following sample defines a default implementation for an interface method:

```
public interface IControl
{
    void Paint() => Console.WriteLine("Default Paint method");
}
public class SampleClass : IControl
{
    // Paint() is inherited from IControl.
}
```

The following sample invokes the default implementation:

```
var sample = new SampleClass();
//sample.Paint();// "Paint" isn't accessible.
var control = sample as IControl;
control.Paint();
```

Any class that implements the `IControl` interface can override the default `Paint` method, either as a public method, or as an explicit interface implementation.

See also

- [C# Programming Guide](#)
- [Object oriented programming](#)
- [Interfaces](#)
- [Inheritance](#)

How to explicitly implement interface members (C# Programming Guide)

12/28/2021 • 2 minutes to read • [Edit Online](#)

This example declares an `interface`, `IDimensions`, and a class, `Box`, which explicitly implements the interface members `GetLength` and `GetWidth`. The members are accessed through the interface instance `dimensions`.

Example

```

interface IDimensions
{
    float GetLength();
    float GetWidth();
}

class Box : IDimensions
{
    float lengthInches;
    float widthInches;

    Box(float length, float width)
    {
        lengthInches = length;
        widthInches = width;
    }
    // Explicit interface member implementation:
    float IDimensions.GetLength()
    {
        return lengthInches;
    }
    // Explicit interface member implementation:
    float IDimensions.GetWidth()
    {
        return widthInches;
    }

    static void Main()
    {
        // Declare a class instance box1:
        Box box1 = new Box(30.0f, 20.0f);

        // Declare an interface instance dimensions:
        IDimensions dimensions = box1;

        // The following commented lines would produce compilation
        // errors because they try to access an explicitly implemented
        // interface member from a class instance:
        //System.Console.WriteLine("Length: {0}", box1.GetLength());
        //System.Console.WriteLine("Width: {0}", box1.GetWidth());

        // Print out the dimensions of the box by calling the methods
        // from an instance of the interface:
        System.Console.WriteLine("Length: {0}", dimensions.GetLength());
        System.Console.WriteLine("Width: {0}", dimensions.GetWidth());
    }
}
/* Output:
    Length: 30
    Width: 20
*/

```

Robust Programming

- Notice that the following lines, in the `Main` method, are commented out because they would produce compilation errors. An interface member that is explicitly implemented cannot be accessed from a [class](#) instance:

```

//System.Console.WriteLine("Length: {0}", box1.GetLength());
//System.Console.WriteLine("Width: {0}", box1.GetWidth());

```

- Notice also that the following lines, in the `Main` method, successfully print out the dimensions of the box because the methods are being called from an instance of the interface:

```
System.Console.WriteLine("Length: {0}", dimensions.GetLength());  
System.Console.WriteLine("Width: {0}", dimensions.GetWidth());
```

See also

- [C# Programming Guide](#)
- [Object oriented programming](#)
- [Interfaces](#)
- [How to explicitly implement members of two interfaces](#)

How to explicitly implement members of two interfaces (C# Programming Guide)

12/28/2021 • 2 minutes to read • [Edit Online](#)

Explicit [interface](#) implementation also allows the programmer to implement two interfaces that have the same member names and give each interface member a separate implementation. This example displays the dimensions of a box in both metric and English units. The Box [class](#) implements two interfaces `IEnglishDimensions` and `IMetricDimensions`, which represent the different measurement systems. Both interfaces have identical member names, `Length` and `Width`.

Example


```

// Declare the English units interface:
interface IEnglishDimensions
{
    float Length();
    float Width();
}

// Declare the metric units interface:
interface IMetricDimensions
{
    float Length();
    float Width();
}

// Declare the Box class that implements the two interfaces:
// IEnglishDimensions and IMetricDimensions:
class Box : IEnglishDimensions, IMetricDimensions
{
    float lengthInches;
    float widthInches;

    public Box(float lengthInches, float widthInches)
    {
        this.lengthInches = lengthInches;
        this.widthInches = widthInches;
    }

    // Explicitly implement the members of IEnglishDimensions:
    float IEnglishDimensions.Length() => lengthInches;

    float IEnglishDimensions.Width() => widthInches;

    // Explicitly implement the members of IMetricDimensions:
    float IMetricDimensions.Length() => lengthInches * 2.54f;

    float IMetricDimensions.Width() => widthInches * 2.54f;

    static void Main()
    {
        // Declare a class instance box1:
        Box box1 = new Box(30.0f, 20.0f);

        // Declare an instance of the English units interface:
        IEnglishDimensions eDimensions = box1;

        // Declare an instance of the metric units interface:
        IMetricDimensions mDimensions = box1;

        // Print dimensions in English units:
        System.Console.WriteLine("Length(in): {0}", eDimensions.Length());
        System.Console.WriteLine("Width (in): {0}", eDimensions.Width());

        // Print dimensions in metric units:
        System.Console.WriteLine("Length(cm): {0}", mDimensions.Length());
        System.Console.WriteLine("Width (cm): {0}", mDimensions.Width());
    }
}

/* Output:
    Length(in): 30
    Width (in): 20
    Length(cm): 76.2
    Width (cm): 50.8
*/

```

If you want to make the default measurements in English units, implement the methods `Length` and `Width` normally, and explicitly implement the `Length` and `Width` methods from the `IMetricDimensions` interface:

```
// Normal implementation:
public float Length() => lengthInches;
public float Width() => widthInches;

// Explicit implementation:
float IMetricDimensions.Length() => lengthInches * 2.54f;
float IMetricDimensions.Width() => widthInches * 2.54f;
```

In this case, you can access the English units from the class instance and access the metric units from the interface instance:

```
public static void Test()
{
    Box box1 = new Box(30.0f, 20.0f);
    IMetricDimensions mDimensions = box1;

    System.Console.WriteLine("Length(in): {0}", box1.Length());
    System.Console.WriteLine("Width (in): {0}", box1.Width());
    System.Console.WriteLine("Length(cm): {0}", mDimensions.Length());
    System.Console.WriteLine("Width (cm): {0}", mDimensions.Width());
}
```

See also

- [C# Programming Guide](#)
- [Object oriented programming](#)
- [Interfaces](#)
- [How to explicitly implement interface members](#)

Delegates (C# Programming Guide)

12/28/2021 • 2 minutes to read • [Edit Online](#)

A [delegate](#) is a type that represents references to methods with a particular parameter list and return type. When you instantiate a delegate, you can associate its instance with any method with a compatible signature and return type. You can invoke (or call) the method through the delegate instance.

Delegates are used to pass methods as arguments to other methods. Event handlers are nothing more than methods that are invoked through delegates. You create a custom method, and a class such as a windows control can call your method when a certain event occurs. The following example shows a delegate declaration:

```
public delegate int PerformCalculation(int x, int y);
```

Any method from any accessible class or struct that matches the delegate type can be assigned to the delegate. The method can be either static or an instance method. This flexibility means you can programmatically change method calls, or plug new code into existing classes.

NOTE

In the context of method overloading, the signature of a method does not include the return value. But in the context of delegates, the signature does include the return value. In other words, a method must have the same return type as the delegate.

This ability to refer to a method as a parameter makes delegates ideal for defining callback methods. You can write a method that compares two objects in your application. That method can be used in a delegate for a sort algorithm. Because the comparison code is separate from the library, the sort method can be more general.

[Function pointers](#) were added to C# 9 for similar scenarios, where you need more control over the calling convention. The code associated with a delegate is invoked using a virtual method added to a delegate type. Using function pointers, you can specify different conventions.

Delegates Overview

Delegates have the following properties:

- Delegates are similar to C++ function pointers, but delegates are fully object-oriented, and unlike C++ pointers to member functions, delegates encapsulate both an object instance and a method.
- Delegates allow methods to be passed as parameters.
- Delegates can be used to define callback methods.
- Delegates can be chained together; for example, multiple methods can be called on a single event.
- Methods don't have to match the delegate type exactly. For more information, see [Using Variance in Delegates](#).
- Lambda expressions are a more concise way of writing inline code blocks. Lambda expressions (in certain contexts) are compiled to delegate types. For more information about lambda expressions, see [Lambda expressions](#).

In This Section

- [Using Delegates](#)

- [When to Use Delegates Instead of Interfaces \(C# Programming Guide\)](#)
- [Delegates with Named vs. Anonymous Methods](#)
- [Using Variance in Delegates](#)
- [How to combine delegates \(Multicast Delegates\)](#)
- [How to declare, instantiate, and use a delegate](#)

C# Language Specification

For more information, see [Delegates](#) in the [C# Language Specification](#). The language specification is the definitive source for C# syntax and usage.

Featured Book Chapters

- [Delegates, Events, and Lambda Expressions](#) in [C# 3.0 Cookbook, Third Edition: More than 250 solutions for C# 3.0 programmers](#)
- [Delegates and Events](#) in [Learning C# 3.0: Fundamentals of C# 3.0](#)

See also

- [Delegate](#)
- [C# Programming Guide](#)
- [Events](#)

Using Delegates (C# Programming Guide)

12/28/2021 • 5 minutes to read • [Edit Online](#)

A [delegate](#) is a type that safely encapsulates a method, similar to a function pointer in C and C++. Unlike C function pointers, delegates are object-oriented, type safe, and secure. The type of a delegate is defined by the name of the delegate. The following example declares a delegate named `Del` that can encapsulate a method that takes a [string](#) as an argument and returns [void](#):

```
public delegate void Del(string message);
```

A delegate object is normally constructed by providing the name of the method the delegate will wrap, or with a [lambda expression](#). Once a delegate is instantiated, a method call made to the delegate will be passed by the delegate to that method. The parameters passed to the delegate by the caller are passed to the method, and the return value, if any, from the method is returned to the caller by the delegate. This is known as invoking the delegate. An instantiated delegate can be invoked as if it were the wrapped method itself. For example:

```
// Create a method for a delegate.
public static void DelegateMethod(string message)
{
    Console.WriteLine(message);
}
```

```
// Instantiate the delegate.
Del handler = DelegateMethod;

// Call the delegate.
handler("Hello World");
```

Delegate types are derived from the [Delegate](#) class in .NET. Delegate types are [sealed](#)—they cannot be derived from—and it is not possible to derive custom classes from [Delegate](#). Because the instantiated delegate is an object, it can be passed as a parameter, or assigned to a property. This allows a method to accept a delegate as a parameter, and call the delegate at some later time. This is known as an asynchronous callback, and is a common method of notifying a caller when a long process has completed. When a delegate is used in this fashion, the code using the delegate does not need any knowledge of the implementation of the method being used. The functionality is similar to the encapsulation interfaces provide.

Another common use of callbacks is defining a custom comparison method and passing that delegate to a sort method. It allows the caller's code to become part of the sort algorithm. The following example method uses the `Del` type as a parameter:

```
public static void MethodWithCallback(int param1, int param2, Del callback)
{
    callback("The number is: " + (param1 + param2).ToString());
}
```

You can then pass the delegate created above to that method:

```
MethodWithCallback(1, 2, handler);
```

and receive the following output to the console:

```
The number is: 3
```

Using the delegate as an abstraction, `MethodWithCallback` does not need to call the console directly—it does not have to be designed with a console in mind. What `MethodWithCallback` does is simply prepare a string and pass the string to another method. This is especially powerful since a delegated method can use any number of parameters.

When a delegate is constructed to wrap an instance method, the delegate references both the instance and the method. A delegate has no knowledge of the instance type aside from the method it wraps, so a delegate can refer to any type of object as long as there is a method on that object that matches the delegate signature. When a delegate is constructed to wrap a static method, it only references the method. Consider the following declarations:

```
public class MethodClass
{
    public void Method1(string message) { }
    public void Method2(string message) { }
}
```

Along with the static `DelegateMethod` shown previously, we now have three methods that can be wrapped by a `Del` instance.

A delegate can call more than one method when invoked. This is referred to as multicasting. To add an extra method to the delegate's list of methods—the invocation list—simply requires adding two delegates using the addition or addition assignment operators ('+' or '+='). For example:

```
var obj = new MethodClass();
Del d1 = obj.Method1;
Del d2 = obj.Method2;
Del d3 = DelegateMethod;

//Both types of assignment are valid.
Del allMethodsDelegate = d1 + d2;
allMethodsDelegate += d3;
```

At this point `allMethodsDelegate` contains three methods in its invocation list—`Method1`, `Method2`, and `DelegateMethod`. The original three delegates, `d1`, `d2`, and `d3`, remain unchanged. When `allMethodsDelegate` is invoked, all three methods are called in order. If the delegate uses reference parameters, the reference is passed sequentially to each of the three methods in turn, and any changes by one method are visible to the next method. When any of the methods throws an exception that is not caught within the method, that exception is passed to the caller of the delegate and no subsequent methods in the invocation list are called. If the delegate has a return value and/or out parameters, it returns the return value and parameters of the last method invoked. To remove a method from the invocation list, use the [subtraction or subtraction assignment operators](#) (`-` or `-=`). For example:

```
//remove Method1
allMethodsDelegate -= d1;

// copy AllMethodsDelegate while removing d2
Del oneMethodDelegate = allMethodsDelegate - d2;
```

Because delegate types are derived from `System.Delegate`, the methods and properties defined by that class can

be called on the delegate. For example, to find the number of methods in a delegate's invocation list, you may write:

```
int invocationCount = d1.GetInvocationList().GetLength(0);
```

Delegates with more than one method in their invocation list derive from [MulticastDelegate](#), which is a subclass of `System.Delegate`. The above code works in either case because both classes support `GetInvocationList`.

Multicast delegates are used extensively in event handling. Event source objects send event notifications to recipient objects that have registered to receive that event. To register for an event, the recipient creates a method designed to handle the event, then creates a delegate for that method and passes the delegate to the event source. The source calls the delegate when the event occurs. The delegate then calls the event handling method on the recipient, delivering the event data. The delegate type for a given event is defined by the event source. For more, see [Events](#).

Comparing delegates of two different types assigned at compile-time will result in a compilation error. If the delegate instances are statically of the type `System.Delegate`, then the comparison is allowed, but will return false at run time. For example:

```
delegate void Delegate1();
delegate void Delegate2();

static void method(Delegate1 d, Delegate2 e, System.Delegate f)
{
    // Compile-time error.
    //Console.WriteLine(d == e);

    // OK at compile-time. False if the run-time type of f
    // is not the same as that of d.
    Console.WriteLine(d == f);
}
```

See also

- [C# Programming Guide](#)
- [Delegates](#)
- [Using Variance in Delegates](#)
- [Variance in Delegates](#)
- [Using Variance for Func and Action Generic Delegates](#)
- [Events](#)

Delegates with Named vs. Anonymous Methods (C# Programming Guide)

12/28/2021 • 2 minutes to read • [Edit Online](#)

A [delegate](#) can be associated with a named method. When you instantiate a delegate by using a named method, the method is passed as a parameter, for example:

```
// Declare a delegate.
delegate void Del(int x);

// Define a named method.
void DoWork(int k) { /* ... */ }

// Instantiate the delegate using the method as a parameter.
Del d = obj.DoWork;
```

This is called using a named method. Delegates constructed with a named method can encapsulate either a [static](#) method or an instance method. Named methods are the only way to instantiate a delegate in earlier versions of C#. However, in a situation where creating a new method is unwanted overhead, C# enables you to instantiate a delegate and immediately specify a code block that the delegate will process when it is called. The block can contain either a [lambda expression](#) or an [anonymous method](#).

The method that you pass as a delegate parameter must have the same signature as the delegate declaration. A delegate instance may encapsulate either static or instance method.

NOTE

Although the delegate can use an [out](#) parameter, we do not recommend its use with multicast event delegates because you cannot know which delegate will be called.

Beginning with C# 10, method groups with a single overload have a *natural type*. This means the compiler can infer the return type and parameter types for the delegate type:

```
var read = Console.Read; // Just one overload; Func<int> inferred
var write = Console.Write; // ERROR: Multiple overloads, can't choose
```

Examples

The following is a simple example of declaring and using a delegate. Notice that both the delegate, `Del`, and the associated method, `MultiplyNumbers`, have the same signature


```

// Declare a delegate
delegate void Del(int i, double j);

class MathClass
{
    static void Main()
    {
        MathClass m = new MathClass();

        // Delegate instantiation using "MultiplyNumbers"
        Del d = m.MultiplyNumbers;

        // Invoke the delegate object.
        Console.WriteLine("Invoking the delegate using 'MultiplyNumbers':");
        for (int i = 1; i <= 5; i++)
        {
            d(i, 2);
        }

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }

    // Declare the associated method.
    void MultiplyNumbers(int m, double n)
    {
        Console.Write(m * n + " ");
    }
}
/* Output:
    Invoking the delegate using 'MultiplyNumbers':
    2 4 6 8 10
*/

```

In the following example, one delegate is mapped to both static and instance methods and returns specific information from each.

```

// Declare a delegate
delegate void Del();

class SampleClass
{
    public void InstanceMethod()
    {
        Console.WriteLine("A message from the instance method.");
    }

    static public void StaticMethod()
    {
        Console.WriteLine("A message from the static method.");
    }
}

class TestSampleClass
{
    static void Main()
    {
        var sc = new SampleClass();

        // Map the delegate to the instance method:
        Del d = sc.InstanceMethod;
        d();

        // Map to the static method:
        d = SampleClass.StaticMethod;
        d();
    }
}
/* Output:
    A message from the instance method.
    A message from the static method.
*/

```

See also

- [C# Programming Guide](#)
- [Delegates](#)
- [How to combine delegates \(Multicast Delegates\)](#)
- [Events](#)

How to combine delegates (Multicast Delegates) (C# Programming Guide)

12/28/2021 • 2 minutes to read • [Edit Online](#)

This example demonstrates how to create multicast delegates. A useful property of `delegate` objects is that multiple objects can be assigned to one delegate instance by using the `+` operator. The multicast delegate contains a list of the assigned delegates. When the multicast delegate is called, it invokes the delegates in the list, in order. Only delegates of the same type can be combined.

The `-` operator can be used to remove a component delegate from a multicast delegate.

Example

```

using System;

// Define a custom delegate that has a string parameter and returns void.
delegate void CustomDel(string s);

class TestClass
{
    // Define two methods that have the same signature as CustomDel.
    static void Hello(string s)
    {
        Console.WriteLine($" Hello, {s}!");
    }

    static void Goodbye(string s)
    {
        Console.WriteLine($" Goodbye, {s}!");
    }

    static void Main()
    {
        // Declare instances of the custom delegate.
        CustomDel hiDel, byeDel, multiDel, multiMinusHiDel;

        // In this example, you can omit the custom delegate if you
        // want to and use Action<string> instead.
        //Action<string> hiDel, byeDel, multiDel, multiMinusHiDel;

        // Create the delegate object hiDel that references the
        // method Hello.
        hiDel = Hello;

        // Create the delegate object byeDel that references the
        // method Goodbye.
        byeDel = Goodbye;

        // The two delegates, hiDel and byeDel, are combined to
        // form multiDel.
        multiDel = hiDel + byeDel;

        // Remove hiDel from the multicast delegate, leaving byeDel,
        // which calls only the method Goodbye.
        multiMinusHiDel = multiDel - hiDel;

        Console.WriteLine("Invoking delegate hiDel:");
        hiDel("A");
        Console.WriteLine("Invoking delegate byeDel:");
        byeDel("B");
        Console.WriteLine("Invoking delegate multiDel:");
        multiDel("C");
        Console.WriteLine("Invoking delegate multiMinusHiDel:");
        multiMinusHiDel("D");
    }
}

/* Output:
Invoking delegate hiDel:
Hello, A!
Invoking delegate byeDel:
Goodbye, B!
Invoking delegate multiDel:
Hello, C!
Goodbye, C!
Invoking delegate multiMinusHiDel:
Goodbye, D!
*/

```

See also

- [MulticastDelegate](#)
- [C# Programming Guide](#)
- [Events](#)

How to declare, instantiate, and use a Delegate (C# Programming Guide)

12/28/2021 • 4 minutes to read • [Edit Online](#)

In C# 1.0 and later, delegates can be declared as shown in the following example.

```
// Declare a delegate.
delegate void Del(string str);

// Declare a method with the same signature as the delegate.
static void Notify(string name)
{
    Console.WriteLine($"Notification received for: {name}");
}
```

```
// Create an instance of the delegate.
Del del1 = new Del(Notify);
```

C# 2.0 provides a simpler way to write the previous declaration, as shown in the following example.

```
// C# 2.0 provides a simpler way to declare an instance of Del.
Del del2 = Notify;
```

In C# 2.0 and later, it is also possible to use an anonymous method to declare and initialize a [delegate](#), as shown in the following example.

```
// Instantiate Del by using an anonymous method.
Del del3 = delegate(string name)
{ Console.WriteLine($"Notification received for: {name}"); };
```

In C# 3.0 and later, delegates can also be declared and instantiated by using a lambda expression, as shown in the following example.

```
// Instantiate Del by using a lambda expression.
Del del4 = name => { Console.WriteLine($"Notification received for: {name}"); };
```

For more information, see [Lambda Expressions](#).

The following example illustrates declaring, instantiating, and using a delegate. The `BookDB` class encapsulates a bookstore database that maintains a database of books. It exposes a method, `ProcessPaperbackBooks`, which finds all paperback books in the database and calls a delegate for each one. The `delegate` type that is used is named `ProcessBookCallback`. The `Test` class uses this class to print the titles and average price of the paperback books.

The use of delegates promotes good separation of functionality between the bookstore database and the client code. The client code has no knowledge of how the books are stored or how the bookstore code finds paperback books. The bookstore code has no knowledge of what processing is performed on the paperback books after it finds them.

Example

```
// A set of classes for handling a bookstore:
namespace Bookstore
{
    using System.Collections;

    // Describes a book in the book list:
    public struct Book
    {
        public string Title;           // Title of the book.
        public string Author;          // Author of the book.
        public decimal Price;          // Price of the book.
        public bool Paperback;          // Is it paperback?

        public Book(string title, string author, decimal price, bool paperBack)
        {
            Title = title;
            Author = author;
            Price = price;
            Paperback = paperBack;
        }
    }

    // Declare a delegate type for processing a book:
    public delegate void ProcessBookCallback(Book book);

    // Maintains a book database.
    public class BookDB
    {
        // List of all books in the database:
        ArrayList list = new ArrayList();

        // Add a book to the database:
        public void AddBook(string title, string author, decimal price, bool paperBack)
        {
            list.Add(new Book(title, author, price, paperBack));
        }

        // Call a passed-in delegate on each paperback book to process it:
        public void ProcessPaperbackBooks(ProcessBookCallback processBook)
        {
            foreach (Book b in list)
            {
                if (b.Paperback)
                {
                    // Calling the delegate:
                    processBook(b);
                }
            }
        }
    }
}

// Using the Bookstore classes:
namespace BookTestClient
{
    using Bookstore;

    // Class to total and average prices of books:
    class PriceTallier
    {
        int countBooks = 0;
        decimal priceBooks = 0.0m;

        internal void AddBookToTotal(Book book)
        {
            countBooks += 1;
            priceBooks += book.Price;
        }
    }
}
```

```

    }

    internal decimal AveragePrice()
    {
        return priceBooks / countBooks;
    }
}

// Class to test the book database:
class Test
{
    // Print the title of the book.
    static void PrintTitle(Book b)
    {
        Console.WriteLine($"    {b.Title}");
    }

    // Execution starts here.
    static void Main()
    {
        BookDB bookDB = new BookDB();

        // Initialize the database with some books:
        AddBooks(bookDB);

        // Print all the titles of paperbacks:
        Console.WriteLine("Paperback Book Titles:");

        // Create a new delegate object associated with the static
        // method Test.PrintTitle:
        bookDB.ProcessPaperbackBooks(PrintTitle);

        // Get the average price of a paperback by using
        // a PriceTallier object:
        PriceTallier tallier = new PriceTallier();

        // Create a new delegate object associated with the nonstatic
        // method AddBookToTotal on the object tallier:
        bookDB.ProcessPaperbackBooks(tallier.AddBookToTotal);

        Console.WriteLine("Average Paperback Book Price: ${0:0.##}",
            tallier.AveragePrice());
    }

    // Initialize the book database with some test books:
    static void AddBooks(BookDB bookDB)
    {
        bookDB.AddBook("The C Programming Language", "Brian W. Kernighan and Dennis M. Ritchie", 19.95m,
true);
        bookDB.AddBook("The Unicode Standard 2.0", "The Unicode Consortium", 39.95m, true);
        bookDB.AddBook("The MS-DOS Encyclopedia", "Ray Duncan", 129.95m, false);
        bookDB.AddBook("Dogbert's Clues for the Clueless", "Scott Adams", 12.00m, true);
    }
}

/* Output:
Paperback Book Titles:
    The C Programming Language
    The Unicode Standard 2.0
    Dogbert's Clues for the Clueless
Average Paperback Book Price: $23.97
*/

```

Robust Programming

- Declaring a delegate.

The following statement declares a new delegate type.

```
public delegate void ProcessBookCallback(Book book);
```

Each delegate type describes the number and types of the arguments, and the type of the return value of methods that it can encapsulate. Whenever a new set of argument types or return value type is needed, a new delegate type must be declared.

- Instantiating a delegate.

After a delegate type has been declared, a delegate object must be created and associated with a particular method. In the previous example, you do this by passing the `PrintTitle` method to the `ProcessPaperbackBooks` method as in the following example:

```
bookDB.ProcessPaperbackBooks(PrintTitle);
```

This creates a new delegate object associated with the [static](#) method `Test.PrintTitle`. Similarly, the non-static method `AddBookToTotal` on the object `totaller` is passed as in the following example:

```
bookDB.ProcessPaperbackBooks(totaller.AddBookToTotal);
```

In both cases a new delegate object is passed to the `ProcessPaperbackBooks` method.

After a delegate is created, the method it is associated with never changes; delegate objects are immutable.

- Calling a delegate.

After a delegate object is created, the delegate object is typically passed to other code that will call the delegate. A delegate object is called by using the name of the delegate object, followed by the parenthesized arguments to be passed to the delegate. Following is an example of a delegate call:

```
processBook(b);
```

A delegate can be either called synchronously, as in this example, or asynchronously by using `BeginInvoke` and `EndInvoke` methods.

See also

- [C# Programming Guide](#)
- [Events](#)
- [Delegates](#)

Arrays (C# Programming Guide)

12/28/2021 • 3 minutes to read • [Edit Online](#)

You can store multiple variables of the same type in an array data structure. You declare an array by specifying the type of its elements. If you want the array to store elements of any type, you can specify `object` as its type. In the unified type system of C#, all types, predefined and user-defined, reference types and value types, inherit directly or indirectly from `Object`.

```
type[] arrayName;
```

Example

The following example creates single-dimensional, multidimensional, and jagged arrays:

```
class TestArraysClass
{
    static void Main()
    {
        // Declare a single-dimensional array of 5 integers.
        int[] array1 = new int[5];

        // Declare and set array element values.
        int[] array2 = new int[] { 1, 3, 5, 7, 9 };

        // Alternative syntax.
        int[] array3 = { 1, 2, 3, 4, 5, 6 };

        // Declare a two dimensional array.
        int[,] multiDimensionalArray1 = new int[2, 3];

        // Declare and set array element values.
        int[,] multiDimensionalArray2 = { { 1, 2, 3 }, { 4, 5, 6 } };

        // Declare a jagged array.
        int[][] jaggedArray = new int[6][];

        // Set the values of the first array in the jagged array structure.
        jaggedArray[0] = new int[4] { 1, 2, 3, 4 };
    }
}
```

Array overview

An array has the following properties:

- An array can be [single-dimensional](#), [multidimensional](#) or [jagged](#).
- The number of dimensions and the length of each dimension are established when the array instance is created. These values can't be changed during the lifetime of the instance.
- The default values of numeric array elements are set to zero, and reference elements are set to `null`.
- A jagged array is an array of arrays, and therefore its elements are reference types and are initialized to `null`.
- Arrays are zero indexed: an array with `n` elements is indexed from `0` to `n-1`.
- Array elements can be of any type, including an array type.

- Array types are [reference types](#) derived from the abstract base type [Array](#). All arrays implement [IList](#), and [IEnumerable](#). You can use the [foreach](#) statement to iterate through an array. Single-dimensional arrays also implement [IList<T>](#) and [IEnumerable<T>](#).

Default value behaviour

- For value types, the array elements are initialized with the [default value](#), the 0-bit pattern; the elements will have the value `0`.
- All the reference types (including the [non-nullable](#)), have the values `null`.
- For nullable value types, `HasValue` is set to `false` and the elements would be set to `null`.

Arrays as Objects

In C#, arrays are actually objects, and not just addressable regions of contiguous memory as in C and C++.

[Array](#) is the abstract base type of all array types. You can use the properties and other class members that [Array](#) has. An example of this is using the [Length](#) property to get the length of an array. The following code assigns the length of the `numbers` array, which is `5`, to a variable called `lengthOfNumbers`:

```
int[] numbers = { 1, 2, 3, 4, 5 };
int lengthOfNumbers = numbers.Length;
```

The [Array](#) class provides many other useful methods and properties for sorting, searching, and copying arrays. The following example uses the [Rank](#) property to display the number of dimensions of an array.

```
class TestArraysClass
{
    static void Main()
    {
        // Declare and initialize an array.
        int[,] theArray = new int[5, 10];
        System.Console.WriteLine("The array has {0} dimensions.", theArray.Rank);
    }
}
// Output: The array has 2 dimensions.
```

See also

- [How to use single-dimensional arrays](#)
- [How to use multi-dimensional arrays](#)
- [How to use jagged arrays](#)
- [Using foreach with arrays](#)
- [Passing arrays as arguments](#)
- [Implicitly typed arrays](#)
- [C# Programming Guide](#)
- [Collections](#)

For more information, see the [C# Language Specification](#). The language specification is the definitive source for C# syntax and usage.

Single-Dimensional Arrays (C# Programming Guide)

12/28/2021 • 2 minutes to read • [Edit Online](#)

You create a single-dimensional array using the `new` operator specifying the array element type and the number of elements. The following example declares an array of five integers:

```
int[] array = new int[5];
```

This array contains the elements from `array[0]` to `array[4]`. The elements of the array are initialized to the **default value** of the element type, `0` for integers.

Arrays can store any element type you specify, such as the following example that declares an array of strings:

```
string[] stringArray = new string[6];
```

Array Initialization

You can initialize the elements of an array when you declare the array. The length specifier isn't needed because it's inferred by the number of elements in the initialization list. For example:

```
int[] array1 = new int[] { 1, 3, 5, 7, 9 };
```

The following code shows a declaration of a string array where each array element is initialized by a name of a day:

```
string[] weekdays = new string[] { "Sun", "Mon", "Tue", "Wed", "Thu", "Fri", "Sat" };
```

You can avoid the `new` expression and the array type when you initialize an array upon declaration, as shown in the following code. This is called an **implicitly typed array**:

```
int[] array2 = { 1, 3, 5, 7, 9 };  
string[] weekdays2 = { "Sun", "Mon", "Tue", "Wed", "Thu", "Fri", "Sat" };
```

You can declare an array variable without creating it, but you must use the `new` operator when you assign a new array to this variable. For example:

```
int[] array3;  
array3 = new int[] { 1, 3, 5, 7, 9 }; // OK  
//array3 = {1, 3, 5, 7, 9}; // Error
```

Value Type and Reference Type Arrays

Consider the following array declaration:

```
SomeType[] array4 = new SomeType[10];
```

The result of this statement depends on whether `SomeType` is a value type or a reference type. If it's a value type, the statement creates an array of 10 elements, each of which has the type `SomeType`. If `SomeType` is a reference type, the statement creates an array of 10 elements, each of which is initialized to a null reference. In both instances, the elements are initialized to the default value for the element type. For more information about value types and reference types, see [Value types](#) and [Reference types](#).

Retrieving data from Array

You can retrieve the data of an array by using an index. For example:

```
string[] weekDays2 = { "Sun", "Mon", "Tue", "Wed", "Thu", "Fri", "Sat" };

Console.WriteLine(weekDays2[0]);
Console.WriteLine(weekDays2[1]);
Console.WriteLine(weekDays2[2]);
Console.WriteLine(weekDays2[3]);
Console.WriteLine(weekDays2[4]);
Console.WriteLine(weekDays2[5]);
Console.WriteLine(weekDays2[6]);

/*Output:
Sun
Mon
Tue
Wed
Thu
Fri
Sat
*/
```

See also

- [Array](#)
- [Arrays](#)
- [Multidimensional Arrays](#)
- [Jagged Arrays](#)

Multidimensional Arrays (C# Programming Guide)

12/28/2021 • 2 minutes to read • [Edit Online](#)

Arrays can have more than one dimension. For example, the following declaration creates a two-dimensional array of four rows and two columns.

```
int[,] array = new int[4, 2];
```

The following declaration creates an array of three dimensions, 4, 2, and 3.

```
int[,,] array1 = new int[4, 2, 3];
```

Array Initialization

You can initialize the array upon declaration, as is shown in the following example.

```
// Two-dimensional array.
int[,] array2D = new int[,] { { 1, 2 }, { 3, 4 }, { 5, 6 }, { 7, 8 } };
// The same array with dimensions specified.
int[,] array2Da = new int[4, 2] { { 1, 2 }, { 3, 4 }, { 5, 6 }, { 7, 8 } };
// A similar array with string elements.
string[,] array2Db = new string[3, 2] { { "one", "two" }, { "three", "four" },
                                         { "five", "six" } };

// Three-dimensional array.
int[,,] array3D = new int[,,] { { { 1, 2, 3 }, { 4, 5, 6 } },
                                { { 7, 8, 9 }, { 10, 11, 12 } } };
// The same array with dimensions specified.
int[,,] array3Da = new int[2, 2, 3] { { { 1, 2, 3 }, { 4, 5, 6 } },
                                       { { 7, 8, 9 }, { 10, 11, 12 } } };

// Accessing array elements.
System.Console.WriteLine(array2D[0, 0]);
System.Console.WriteLine(array2D[0, 1]);
System.Console.WriteLine(array2D[1, 0]);
System.Console.WriteLine(array2D[1, 1]);
System.Console.WriteLine(array2D[3, 0]);
System.Console.WriteLine(array2Db[1, 0]);
System.Console.WriteLine(array3Da[1, 0, 1]);
System.Console.WriteLine(array3D[1, 1, 2]);

// Getting the total count of elements or the length of a given dimension.
var allLength = array3D.Length;
var total = 1;
for (int i = 0; i < array3D.Rank; i++)
{
    total *= array3D.GetLength(i);
}
System.Console.WriteLine("{0} equals {1}", allLength, total);

// Output:
// 1
// 2
// 3
// 4
// 7
// three
// 8
// 12
// 12 equals 12
```

You can also initialize the array without specifying the rank.

```
int[,] array4 = { { 1, 2 }, { 3, 4 }, { 5, 6 }, { 7, 8 } };
```

If you choose to declare an array variable without initialization, you must use the `new` operator to assign an array to the variable. The use of `new` is shown in the following example.

```
int[,] array5;
array5 = new int[,] { { 1, 2 }, { 3, 4 }, { 5, 6 }, { 7, 8 } }; // OK
//array5 = { {1,2}, {3,4}, {5,6}, {7,8} }; // Error
```

The following example assigns a value to a particular array element.

```
array5[2, 1] = 25;
```

Similarly, the following example gets the value of a particular array element and assigns it to variable

```
elementValue .
```

```
int elementValue = array5[2, 1];
```

The following code example initializes the array elements to default values (except for jagged arrays).

```
int[,] array6 = new int[10, 10];
```

See also

- [C# Programming Guide](#)
- [Arrays](#)
- [Single-Dimensional Arrays](#)
- [Jagged Arrays](#)

Jagged Arrays (C# Programming Guide)

12/28/2021 • 3 minutes to read • [Edit Online](#)

A jagged array is an array whose elements are arrays, possibly of different sizes. A jagged array is sometimes called an "array of arrays." The following examples show how to declare, initialize, and access jagged arrays.

The following is a declaration of a single-dimensional array that has three elements, each of which is a single-dimensional array of integers:

```
int[][] jaggedArray = new int[3][];
```

Before you can use `jaggedArray`, its elements must be initialized. You can initialize the elements like this:

```
jaggedArray[0] = new int[5];
jaggedArray[1] = new int[4];
jaggedArray[2] = new int[2];
```

Each of the elements is a single-dimensional array of integers. The first element is an array of 5 integers, the second is an array of 4 integers, and the third is an array of 2 integers.

It is also possible to use initializers to fill the array elements with values, in which case you do not need the array size. For example:

```
jaggedArray[0] = new int[] { 1, 3, 5, 7, 9 };
jaggedArray[1] = new int[] { 0, 2, 4, 6 };
jaggedArray[2] = new int[] { 11, 22 };
```

You can also initialize the array upon declaration like this:

```
int[][] jaggedArray2 = new int[][]
{
    new int[] { 1, 3, 5, 7, 9 },
    new int[] { 0, 2, 4, 6 },
    new int[] { 11, 22 }
};
```

You can use the following shorthand form. Notice that you cannot omit the `new` operator from the elements initialization because there is no default initialization for the elements:

```
int[][] jaggedArray3 =
{
    new int[] { 1, 3, 5, 7, 9 },
    new int[] { 0, 2, 4, 6 },
    new int[] { 11, 22 }
};
```

A jagged array is an array of arrays, and therefore its elements are reference types and are initialized to `null`.

You can access individual array elements like these examples:

```
// Assign 77 to the second element ([1]) of the first array ([0]):  
jaggedArray3[0][1] = 77;  
  
// Assign 88 to the second element ([1]) of the third array ([2]):  
jaggedArray3[2][1] = 88;
```

It's possible to mix jagged and multidimensional arrays. The following is a declaration and initialization of a single-dimensional jagged array that contains three two-dimensional array elements of different sizes. For more information, see [Multidimensional Arrays](#).

```
int[,] jaggedArray4 = new int[3][,]  
{  
    new int[,] { {1,3}, {5,7} },  
    new int[,] { {0,2}, {4,6}, {8,10} },  
    new int[,] { {11,22}, {99,88}, {0,9} }  
};
```

You can access individual elements as shown in this example, which displays the value of the element `[1,0]` of the first array (value `5`):

```
System.Console.WriteLine("{0}", jaggedArray4[0][1, 0]);
```

The method `Length` returns the number of arrays contained in the jagged array. For example, assuming you have declared the previous array, this line:

```
System.Console.WriteLine(jaggedArray4.Length);
```

returns a value of 3.

Example

This example builds an array whose elements are themselves arrays. Each one of the array elements has a different size.

```

class ArrayTest
{
    static void Main()
    {
        // Declare the array of two elements.
        int[][] arr = new int[2][];

        // Initialize the elements.
        arr[0] = new int[5] { 1, 3, 5, 7, 9 };
        arr[1] = new int[4] { 2, 4, 6, 8 };

        // Display the array elements.
        for (int i = 0; i < arr.Length; i++)
        {
            System.Console.Write("Element({0}): ", i);

            for (int j = 0; j < arr[i].Length; j++)
            {
                System.Console.Write("{0}{1}", arr[i][j], j == (arr[i].Length - 1) ? "" : " ");
            }
            System.Console.WriteLine();
        }
        // Keep the console window open in debug mode.
        System.Console.WriteLine("Press any key to exit.");
        System.Console.ReadKey();
    }
}
/* Output:
    Element(0): 1 3 5 7 9
    Element(1): 2 4 6 8
*/

```

See also

- [Array](#)
- [C# Programming Guide](#)
- [Arrays](#)
- [Single-Dimensional Arrays](#)
- [Multidimensional Arrays](#)

Using foreach with arrays (C# Programming Guide)

12/28/2021 • 2 minutes to read • [Edit Online](#)

The `foreach` statement provides a simple, clean way to iterate through the elements of an array.

For single-dimensional arrays, the `foreach` statement processes elements in increasing index order, starting with index 0 and ending with index `Length - 1`:

```
int[] numbers = { 4, 5, 6, 1, 2, 3, -2, -1, 0 };
foreach (int i in numbers)
{
    System.Console.Write("{0} ", i);
}
// Output: 4 5 6 1 2 3 -2 -1 0
```

For multi-dimensional arrays, elements are traversed such that the indices of the rightmost dimension are increased first, then the next left dimension, and so on to the left:

```
int[,] numbers2D = new int[3, 2] { { 9, 99 }, { 3, 33 }, { 5, 55 } };
// Or use the short form:
// int[,] numbers2D = { { 9, 99 }, { 3, 33 }, { 5, 55 } };

foreach (int i in numbers2D)
{
    System.Console.Write("{0} ", i);
}
// Output: 9 99 3 33 5 55
```

However, with multidimensional arrays, using a nested `for` loop gives you more control over the order in which to process the array elements.

See also

- [Array](#)
- [C# Programming Guide](#)
- [Arrays](#)
- [Single-Dimensional Arrays](#)
- [Multidimensional Arrays](#)
- [Jagged Arrays](#)

Passing arrays as arguments (C# Programming Guide)

12/28/2021 • 3 minutes to read • [Edit Online](#)

Arrays can be passed as arguments to method parameters. Because arrays are reference types, the method can change the value of the elements.

Passing single-dimensional arrays as arguments

You can pass an initialized single-dimensional array to a method. For example, the following statement sends an array to a print method.

```
int[] theArray = { 1, 3, 5, 7, 9 };  
PrintArray(theArray);
```

The following code shows a partial implementation of the print method.

```
void PrintArray(int[] arr)  
{  
    // Method code.  
}
```

You can initialize and pass a new array in one step, as is shown in the following example.

```
PrintArray(new int[] { 1, 3, 5, 7, 9 });
```

Example

In the following example, an array of strings is initialized and passed as an argument to a `DisplayArray` method for strings. The method displays the elements of the array. Next, the `ChangeArray` method reverses the array elements, and then the `ChangeArrayElements` method modifies the first three elements of the array. After each method returns, the `DisplayArray` method shows that passing an array by value doesn't prevent changes to the array elements.

```

using System;

class ArrayExample
{
    static void DisplayArray(string[] arr) => Console.WriteLine(string.Join(" ", arr));

    // Change the array by reversing its elements.
    static void ChangeArray(string[] arr) => Array.Reverse(arr);

    static void ChangeArrayElements(string[] arr)
    {
        // Change the value of the first three array elements.
        arr[0] = "Mon";
        arr[1] = "Wed";
        arr[2] = "Fri";
    }

    static void Main()
    {
        // Declare and initialize an array.
        string[] weekDays = { "Sun", "Mon", "Tue", "Wed", "Thu", "Fri", "Sat" };
        // Display the array elements.
        DisplayArray(weekDays);
        Console.WriteLine();

        // Reverse the array.
        ChangeArray(weekDays);
        // Display the array again to verify that it stays reversed.
        Console.WriteLine("Array weekDays after the call to ChangeArray:");
        DisplayArray(weekDays);
        Console.WriteLine();

        // Assign new values to individual array elements.
        ChangeArrayElements(weekDays);
        // Display the array again to verify that it has changed.
        Console.WriteLine("Array weekDays after the call to ChangeArrayElements:");
        DisplayArray(weekDays);
    }
}

// The example displays the following output:
//      Sun Mon Tue Wed Thu Fri Sat
//
//      Array weekDays after the call to ChangeArray:
//      Sat Fri Thu Wed Tue Mon Sun
//
//      Array weekDays after the call to ChangeArrayElements:
//      Mon Wed Fri Wed Tue Mon Sun

```

Passing multidimensional arrays as arguments

You pass an initialized multidimensional array to a method in the same way that you pass a one-dimensional array.

```

int[,] theArray = { { 1, 2 }, { 2, 3 }, { 3, 4 } };
Print2DArray(theArray);

```

The following code shows a partial declaration of a print method that accepts a two-dimensional array as its argument.

```
void Print2DArray(int[,] arr)
{
    // Method code.
}
```

You can initialize and pass a new array in one step, as is shown in the following example:

```
Print2DArray(new int[,] { { 1, 2 }, { 3, 4 }, { 5, 6 }, { 7, 8 } });
```

Example

In the following example, a two-dimensional array of integers is initialized and passed to the `Print2DArray` method. The method displays the elements of the array.

```
class ArrayClass2D
{
    static void Print2DArray(int[,] arr)
    {
        // Display the array elements.
        for (int i = 0; i < arr.GetLength(0); i++)
        {
            for (int j = 0; j < arr.GetLength(1); j++)
            {
                System.Console.WriteLine("Element({0},{1})={2}", i, j, arr[i, j]);
            }
        }
    }
    static void Main()
    {
        // Pass the array as an argument.
        Print2DArray(new int[,] { { 1, 2 }, { 3, 4 }, { 5, 6 }, { 7, 8 } });

        // Keep the console window open in debug mode.
        System.Console.WriteLine("Press any key to exit.");
        System.Console.ReadKey();
    }
}
/* Output:
    Element(0,0)=1
    Element(0,1)=2
    Element(1,0)=3
    Element(1,1)=4
    Element(2,0)=5
    Element(2,1)=6
    Element(3,0)=7
    Element(3,1)=8
*/
```

See also

- [C# Programming Guide](#)
- [Arrays](#)
- [Single-Dimensional Arrays](#)
- [Multidimensional Arrays](#)
- [Jagged Arrays](#)

Implicitly Typed Arrays (C# Programming Guide)

12/28/2021 • 2 minutes to read • [Edit Online](#)

You can create an implicitly-typed array in which the type of the array instance is inferred from the elements specified in the array initializer. The rules for any implicitly-typed variable also apply to implicitly-typed arrays. For more information, see [Implicitly Typed Local Variables](#).

Implicitly-typed arrays are usually used in query expressions together with anonymous types and object and collection initializers.

The following examples show how to create an implicitly-typed array:

```
class ImplicitlyTypedArraySample
{
    static void Main()
    {
        var a = new[] { 1, 10, 100, 1000 }; // int[]
        var b = new[] { "hello", null, "world" }; // string[]

        // single-dimension jagged array
        var c = new[]
        {
            new[] { 1, 2, 3, 4 },
            new[] { 5, 6, 7, 8 }
        };

        // jagged array of strings
        var d = new[]
        {
            new[] { "Luca", "Mads", "Luke", "Dinesh" },
            new[] { "Karen", "Suma", "Frances" }
        };
    }
}
```

In the previous example, notice that with implicitly-typed arrays, no square brackets are used on the left side of the initialization statement. Note also that jagged arrays are initialized by using `new []` just like single-dimension arrays.

Implicitly-typed Arrays in Object Initializers

When you create an anonymous type that contains an array, the array must be implicitly typed in the type's object initializer. In the following example, `contacts` is an implicitly-typed array of anonymous types, each of which contains an array named `PhoneNumbers`. Note that the `var` keyword is not used inside the object initializers.


```
var contacts = new[]  
{  
    new {  
        Name = " Eugene Zabokritski",  
        PhoneNumbers = new[] { "206-555-0108", "425-555-0001" }  
    },  
    new {  
        Name = " Hanying Feng",  
        PhoneNumbers = new[] { "650-555-0199" }  
    }  
};
```

See also

- [C# Programming Guide](#)
- [Implicitly Typed Local Variables](#)
- [Arrays](#)
- [Anonymous Types](#)
- [Object and Collection Initializers](#)
- [var](#)
- [LINQ in C#](#)

Strings (C# Programming Guide)

12/28/2021 • 13 minutes to read • [Edit Online](#)

A string is an object of type [String](#) whose value is text. Internally, the text is stored as a sequential read-only collection of [Char](#) objects. There is no null-terminating character at the end of a C# string; therefore a C# string can contain any number of embedded null characters ('\0'). The [Length](#) property of a string represents the number of [Char](#) objects it contains, not the number of Unicode characters. To access the individual Unicode code points in a string, use the [StringInfo](#) object.

string vs. System.String

In C#, the `string` keyword is an alias for [String](#). Therefore, `String` and `string` are equivalent, regardless it is recommended to use the provided alias `string` as it works even without `using System;`. The `String` class provides many methods for safely creating, manipulating, and comparing strings. In addition, the C# language overloads some operators to simplify common string operations. For more information about the keyword, see [string](#). For more information about the type and its methods, see [String](#).

Declaring and Initializing Strings

You can declare and initialize strings in various ways, as shown in the following example:

```
// Declare without initializing.
string message1;

// Initialize to null.
string message2 = null;

// Initialize as an empty string.
// Use the Empty constant instead of the literal "".
string message3 = System.String.Empty;

// Initialize with a regular string literal.
string oldPath = "c:\\Program Files\\Microsoft Visual Studio 8.0";

// Initialize with a verbatim string literal.
string newPath = @"c:\Program Files\Microsoft Visual Studio 9.0";

// Use System.String if you prefer.
System.String greeting = "Hello World!";

// In local variables (i.e. within a method body)
// you can use implicit typing.
var temp = "I'm still a strongly-typed System.String!";

// Use a const string to prevent 'message4' from
// being used to store another string value.
const string message4 = "You can't get rid of me!";

// Use the String constructor only when creating
// a string from a char*, char[], or sbyte*. See
// System.String documentation for details.
char[] letters = { 'A', 'B', 'C' };
string alphabet = new string(letters);
```

Note that you do not use the [new](#) operator to create a string object except when initializing the string with an array of chars.

Initialize a string with the [Empty](#) constant value to create a new [String](#) object whose string is of zero length. The string literal representation of a zero-length string is `""`. By initializing strings with the [Empty](#) value instead of [null](#), you can reduce the chances of a [NullReferenceException](#) occurring. Use the static [IsNullOrEmpty\(String\)](#) method to verify the value of a string before you try to access it.

Immutability of String Objects

String objects are *immutable*: they cannot be changed after they have been created. All of the [String](#) methods and C# operators that appear to modify a string actually return the results in a new string object. In the following example, when the contents of `s1` and `s2` are concatenated to form a single string, the two original strings are unmodified. The `+=` operator creates a new string that contains the combined contents. That new object is assigned to the variable `s1`, and the original object that was assigned to `s1` is released for garbage collection because no other variable holds a reference to it.

```
string s1 = "A string is more ";
string s2 = "than the sum of its chars.";

// Concatenate s1 and s2. This actually creates a new
// string object and stores it in s1, releasing the
// reference to the original object.
s1 += s2;

System.Console.WriteLine(s1);
// Output: A string is more than the sum of its chars.
```

Because a string "modification" is actually a new string creation, you must use caution when you create references to strings. If you create a reference to a string, and then "modify" the original string, the reference will continue to point to the original object instead of the new object that was created when the string was modified. The following code illustrates this behavior:

```
string s1 = "Hello ";
string s2 = s1;
s1 += "World";

System.Console.WriteLine(s2);
//Output: Hello
```

For more information about how to create new strings that are based on modifications such as search and replace operations on the original string, see [How to modify string contents](#).

Regular and Verbatim String Literals

Use regular string literals when you must embed escape characters provided by C#, as shown in the following example:

```

string columns = "Column 1\tColumn 2\tColumn 3";
//Output: Column 1      Column 2      Column 3

string rows = "Row 1\r\nRow 2\r\nRow 3";
/* Output:
    Row 1
    Row 2
    Row 3
*/

string title = "\"The \u00C6olean Harp\"", by Samuel Taylor Coleridge";
//Output: "The Æolean Harp", by Samuel Taylor Coleridge

```

Use verbatim strings for convenience and better readability when the string text contains backslash characters, for example in file paths. Because verbatim strings preserve new line characters as part of the string text, they can be used to initialize multiline strings. Use double quotation marks to embed a quotation mark inside a verbatim string. The following example shows some common uses for verbatim strings:

```

string filePath = @"C:\Users\scoleridge\Documents\";
//Output: C:\Users\scoleridge\Documents\

string text = @"My pensive SARA ! thy soft cheek reclined
    Thus on mine arm, most soothing sweet it is
    To sit beside our Cot,...";
/* Output:
My pensive SARA ! thy soft cheek reclined
    Thus on mine arm, most soothing sweet it is
    To sit beside our Cot,...
*/

string quote = @"Her name was ""Sara.""";
//Output: Her name was "Sara."

```

String Escape Sequences

ESCAPE SEQUENCE	CHARACTER NAME	UNICODE ENCODING
\'	Single quote	0x0027
\"	Double quote	0x0022
\\	Backslash	0x005C
\0	Null	0x0000
\a	Alert	0x0007
\b	Backspace	0x0008
\f	Form feed	0x000C
\n	New line	0x000A
\r	Carriage return	0x000D
\t	Horizontal tab	0x0009

ESCAPE SEQUENCE	CHARACTER NAME	UNICODE ENCODING
<code>\v</code>	Vertical tab	0x000B
<code>\u</code>	Unicode escape sequence (UTF-16)	<code>\uHHHH</code> (range: 0000 - FFFF; example: <code>\u00E7</code> = "ç")
<code>\U</code>	Unicode escape sequence (UTF-32)	<code>\U00HHHHHH</code> (range: 000000 - 10FFFF; example: <code>\U0001F47D</code> = "🍷")
<code>\x</code>	Unicode escape sequence similar to <code>"\u"</code> except with variable length	<code>\xH[H][H][H]</code> (range: 0 - FFFF; example: <code>\x00E7</code> or <code>\xE7</code> or <code>\xE7</code> = "ç")

WARNING

When using the `\x` escape sequence and specifying less than 4 hex digits, if the characters that immediately follow the escape sequence are valid hex digits (i.e. 0-9, A-F, and a-f), they will be interpreted as being part of the escape sequence. For example, `\xA1` produces "¡", which is code point U+00A1. However, if the next character is "A" or "a", then the escape sequence will instead be interpreted as being `\xA1A` and produce "■", which is code point U+0A1A. In such cases, specifying all 4 hex digits (e.g. `\x00A1`) will prevent any possible misinterpretation.

NOTE

At compile time, verbatim strings are converted to ordinary strings with all the same escape sequences. Therefore, if you view a verbatim string in the debugger watch window, you will see the escape characters that were added by the compiler, not the verbatim version from your source code. For example, the verbatim string `@ "C:\files.txt"` will appear in the watch window as `"C:\\files.txt"`.

Format Strings

A format string is a string whose contents are determined dynamically at run time. Format strings are created by embedding *interpolated expressions* or placeholders inside of braces within a string. Everything inside the braces (`{...}`) will be resolved to a value and output as a formatted string at run time. There are two methods to create format strings: string interpolation and composite formatting.

String Interpolation

Available in C# 6.0 and later, *interpolated strings* are identified by the `$` special character and include interpolated expressions in braces. If you are new to string interpolation, see the [String interpolation - C# interactive tutorial](#) for a quick overview.

Use string interpolation to improve the readability and maintainability of your code. String interpolation achieves the same results as the `String.Format` method, but improves ease of use and inline clarity.

```
var jh = (firstName: "Jupiter", lastName: "Hammon", born: 1711, published: 1761);
Console.WriteLine($"{jh.firstName} {jh.lastName} was an African American poet born in {jh.born}.");
Console.WriteLine($"He was first published in {jh.published} at the age of {jh.published - jh.born}.");
Console.WriteLine($"He'd be over {Math.Round((2018d - jh.born) / 100d) * 100d} years old today.");

// Output:
// Jupiter Hammon was an African American poet born in 1711.
// He was first published in 1761 at the age of 50.
// He'd be over 300 years old today.
```

Beginning with C# 10, you can use string interpolation to initialize a constant string when all the expressions used for placeholders are also constant strings.

Composite Formatting

The [String.Format](#) utilizes placeholders in braces to create a format string. This example results in similar output to the string interpolation method used above.

```
var pw = (firstName: "Phillis", lastName: "Wheatley", born: 1753, published: 1773);
Console.WriteLine("{0} {1} was an African American poet born in {2}.", pw.firstName, pw.lastName, pw.born);
Console.WriteLine("She was first published in {0} at the age of {1}.", pw.published, pw.published - pw.born);
Console.WriteLine("She'd be over {0} years old today.", Math.Round((2018d - pw.born) / 100d) * 100d);

// Output:
// Phillis Wheatley was an African American poet born in 1753.
// She was first published in 1773 at the age of 20.
// She'd be over 300 years old today.
```

For more information on formatting .NET types see [Formatting Types in .NET](#).

Substrings

A substring is any sequence of characters that is contained in a string. Use the [Substring](#) method to create a new string from a part of the original string. You can search for one or more occurrences of a substring by using the [IndexOf](#) method. Use the [Replace](#) method to replace all occurrences of a specified substring with a new string. Like the [Substring](#) method, [Replace](#) actually returns a new string and does not modify the original string. For more information, see [How to search strings](#) and [How to modify string contents](#).

```
string s3 = "Visual C# Express";
System.Console.WriteLine(s3.Substring(7, 2));
// Output: "C#"

System.Console.WriteLine(s3.Replace("C#", "Basic"));
// Output: "Visual Basic Express"

// Index values are zero-based
int index = s3.IndexOf("C");
// index = 7
```

Accessing Individual Characters

You can use array notation with an index value to acquire read-only access to individual characters, as in the following example:

```
string s5 = "Printing backwards";

for (int i = 0; i < s5.Length; i++)
{
    System.Console.Write(s5[s5.Length - i - 1]);
}
// Output: "sdrawkcb gnitnirP"
```

If the [String](#) methods do not provide the functionality that you must have to modify individual characters in a string, you can use a [StringBuilder](#) object to modify the individual chars "in-place", and then create a new string to store the results by using the [StringBuilder](#) methods. In the following example, assume that you must modify the original string in a particular way and then store the results for future use:

```
string question = "hOW DOES mICROSOFT wORD DEAL WITH THE cAPS LOCK KEY?";
System.Text.StringBuilder sb = new System.Text.StringBuilder(question);

for (int j = 0; j < sb.Length; j++)
{
    if (System.Char.IsLower(sb[j]) == true)
        sb[j] = System.Char.ToUpper(sb[j]);
    else if (System.Char.IsUpper(sb[j]) == true)
        sb[j] = System.Char.ToLower(sb[j]);
}
// Store the new string.
string corrected = sb.ToString();
System.Console.WriteLine(corrected);
// Output: How does Microsoft Word deal with the Caps Lock key?
```

Null Strings and Empty Strings

An empty string is an instance of a [System.String](#) object that contains zero characters. Empty strings are used often in various programming scenarios to represent a blank text field. You can call methods on empty strings because they are valid [System.String](#) objects. Empty strings are initialized as follows:

```
string s = String.Empty;
```

By contrast, a null string does not refer to an instance of a [System.String](#) object and any attempt to call a method on a null string causes a [NullReferenceException](#). However, you can use null strings in concatenation and comparison operations with other strings. The following examples illustrate some cases in which a reference to a null string does and does not cause an exception to be thrown:

```

static void Main()
{
    string str = "hello";
    string nullStr = null;
    string emptyStr = String.Empty;

    string tempStr = str + nullStr;
    // Output of the following line: hello
    Console.WriteLine(tempStr);

    bool b = (emptyStr == nullStr);
    // Output of the following line: False
    Console.WriteLine(b);

    // The following line creates a new empty string.
    string newStr = emptyStr + nullStr;

    // Null strings and empty strings behave differently. The following
    // two lines display 0.
    Console.WriteLine(emptyStr.Length);
    Console.WriteLine(newStr.Length);
    // The following line raises a NullReferenceException.
    //Console.WriteLine(nullStr.Length);

    // The null character can be displayed and counted, like other chars.
    string s1 = "\x0" + "abc";
    string s2 = "abc" + "\x0";
    // Output of the following line: * abc*
    Console.WriteLine("'" + s1 + "'");
    // Output of the following line: *abc *
    Console.WriteLine("'" + s2 + "'");
    // Output of the following line: 4
    Console.WriteLine(s2.Length);
}

```

Using StringBuilder for Fast String Creation

String operations in .NET are highly optimized and in most cases do not significantly impact performance. However, in some scenarios such as tight loops that are executing many hundreds or thousands of times, string operations can affect performance. The [StringBuilder](#) class creates a string buffer that offers better performance if your program performs many string manipulations. The [StringBuilder](#) string also enables you to reassign individual characters, something the built-in string data type does not support. This code, for example, changes the content of a string without creating a new string:

```

System.Text.StringBuilder sb = new System.Text.StringBuilder("Rat: the ideal pet");
sb[0] = 'C';
System.Console.WriteLine(sb.ToString());
System.Console.ReadLine();

//Outputs Cat: the ideal pet

```

In this example, a [StringBuilder](#) object is used to create a string from a set of numeric types:


```

using System;
using System.Text;

namespace CSRefStrings
{
    class TestStringBuilder
    {
        static void Main()
        {
            var sb = new StringBuilder();

            // Create a string composed of numbers 0 - 9
            for (int i = 0; i < 10; i++)
            {
                sb.Append(i.ToString());
            }
            Console.WriteLine(sb); // displays 0123456789

            // Copy one character of the string (not possible with a System.String)
            sb[0] = sb[9];

            Console.WriteLine(sb); // displays 9123456789
            Console.WriteLine();
        }
    }
}

```

Strings, Extension Methods and LINQ

Because the [String](#) type implements [IEnumerable<T>](#), you can use the extension methods defined in the [Enumerable](#) class on strings. To avoid visual clutter, these methods are excluded from IntelliSense for the [String](#) type, but they are available nevertheless. You can also use LINQ query expressions on strings. For more information, see [LINQ and Strings](#).

Related Topics

TOPIC	DESCRIPTION
How to modify string contents	Illustrates techniques to transform strings and modify the contents of strings.
How to compare strings	Shows how to perform ordinal and culture specific comparisons of strings.
How to concatenate multiple strings	Demonstrates various ways to join multiple strings into one.
How to parse strings using String.Split	Contains code examples that illustrate how to use the <code>String.Split</code> method to parse strings.
How to search strings	Explains how to use search for specific text or patterns in strings.
How to determine whether a string represents a numeric value	Shows how to safely parse a string to see whether it has a valid numeric value.
String interpolation	Describes the string interpolation feature that provides a convenient syntax to format strings.

TOPIC	DESCRIPTION
Basic String Operations	Provides links to topics that use System.String and System.Text.StringBuilder methods to perform basic string operations.
Parsing Strings	Describes how to convert string representations of .NET base types to instances of the corresponding types.
Parsing Date and Time Strings in .NET	Shows how to convert a string such as "01/24/2008" to a System.DateTime object.
Comparing Strings	Includes information about how to compare strings and provides examples in C# and Visual Basic.
Using the StringBuilder Class	Describes how to create and modify dynamic string objects by using the StringBuilder class.
LINQ and Strings	Provides information about how to perform various string operations by using LINQ queries.
C# Programming Guide	Provides links to topics that explain programming constructs in C#.

How to determine whether a string represents a numeric value (C# Programming Guide)

12/28/2021 • 2 minutes to read • [Edit Online](#)

To determine whether a string is a valid representation of a specified numeric type, use the static `TryParse` method that is implemented by all primitive numeric types and also by types such as `DateTime` and `IPAddress`. The following example shows how to determine whether "108" is a valid `int`.

```
int i = 0;
string s = "108";
bool result = int.TryParse(s, out i); //i now = 108
```

If the string contains nonnumeric characters or the numeric value is too large or too small for the particular type you have specified, `TryParse` returns false and sets the out parameter to zero. Otherwise, it returns true and sets the out parameter to the numeric value of the string.

NOTE

A string may contain only numeric characters and still not be valid for the type whose `TryParse` method that you use. For example, "256" is not a valid value for `byte` but it is valid for `int`. "98.6" is not a valid value for `int` but it is a valid `decimal`.

Example

The following examples show how to use `TryParse` with string representations of `long`, `byte`, and `decimal` values.

```
string numString = "1287543"; //"1287543.0" will return false for a long
long number1 = 0;
bool canConvert = long.TryParse(numString, out number1);
if (canConvert == true)
    Console.WriteLine("number1 now = {0}", number1);
else
    Console.WriteLine("numString is not a valid long");

byte number2 = 0;
numString = "255"; // A value of 256 will return false
canConvert = byte.TryParse(numString, out number2);
if (canConvert == true)
    Console.WriteLine("number2 now = {0}", number2);
else
    Console.WriteLine("numString is not a valid byte");

decimal number3 = 0;
numString = "27.3"; //"27" is also a valid decimal
canConvert = decimal.TryParse(numString, out number3);
if (canConvert == true)
    Console.WriteLine("number3 now = {0}", number3);
else
    Console.WriteLine("number3 is not a valid decimal");
```

Robust Programming

Primitive numeric types also implement the `Parse` static method, which throws an exception if the string is not a valid number. `TryParse` is generally more efficient because it just returns false if the number is not valid.

.NET Security

Always use the `TryParse` or `Parse` methods to validate user input from controls such as text boxes and combo boxes.

See also

- [How to convert a byte array to an int](#)
- [How to convert a string to a number](#)
- [How to convert between hexadecimal strings and numeric types](#)
- [Parsing Numeric Strings](#)
- [Formatting Types](#)

Indexers (C# Programming Guide)

12/28/2021 • 2 minutes to read • [Edit Online](#)

Indexers allow instances of a class or struct to be indexed just like arrays. The indexed value can be set or retrieved without explicitly specifying a type or instance member. Indexers resemble [properties](#) except that their accessors take parameters.

The following example defines a generic class with simple [get](#) and [set](#) accessor methods to assign and retrieve values. The `Program` class creates an instance of this class for storing strings.

```
using System;

class SampleCollection<T>
{
    // Declare an array to store the data elements.
    private T[] arr = new T[100];

    // Define the indexer to allow client code to use [] notation.
    public T this[int i]
    {
        get { return arr[i]; }
        set { arr[i] = value; }
    }
}

class Program
{
    static void Main()
    {
        var stringCollection = new SampleCollection<string>();
        stringCollection[0] = "Hello, World";
        Console.WriteLine(stringCollection[0]);
    }
}

// The example displays the following output:
//      Hello, World.
```

NOTE

For more examples, see [Related Sections](#).

Expression Body Definitions

It is common for an indexer's get or set accessor to consist of a single statement that either returns or sets a value. Expression-bodied members provide a simplified syntax to support this scenario. Starting with C# 6, a read-only indexer can be implemented as an expression-bodied member, as the following example shows.

```

using System;

class SampleCollection<T>
{
    // Declare an array to store the data elements.
    private T[] arr = new T[100];
    int nextIndex = 0;

    // Define the indexer to allow client code to use [] notation.
    public T this[int i] => arr[i];

    public void Add(T value)
    {
        if (nextIndex >= arr.Length)
            throw new IndexOutOfRangeException($"The collection can hold only {arr.Length} elements.");
        arr[nextIndex++] = value;
    }
}

class Program
{
    static void Main()
    {
        var stringCollection = new SampleCollection<string>();
        stringCollection.Add("Hello, World");
        System.Console.WriteLine(stringCollection[0]);
    }
}
// The example displays the following output:
//      Hello, World.

```

Note that `=>` introduces the expression body, and that the `get` keyword is not used.

Starting with C# 7.0, both the get and set accessor can be implemented as expression-bodied members. In this case, both `get` and `set` keywords must be used. For example:

```

using System;

class SampleCollection<T>
{
    // Declare an array to store the data elements.
    private T[] arr = new T[100];

    // Define the indexer to allow client code to use [] notation.
    public T this[int i]
    {
        get => arr[i];
        set => arr[i] = value;
    }
}

class Program
{
    static void Main()
    {
        var stringCollection = new SampleCollection<string>();
        stringCollection[0] = "Hello, World.";
        Console.WriteLine(stringCollection[0]);
    }
}
// The example displays the following output:
//      Hello, World.

```

Indexers Overview

- Indexers enable objects to be indexed in a similar manner to arrays.
- A `get` accessor returns a value. A `set` accessor assigns a value.
- The `this` keyword is used to define the indexer.
- The `value` keyword is used to define the value being assigned by the `set` accessor.
- Indexers do not have to be indexed by an integer value; it is up to you how to define the specific look-up mechanism.
- Indexers can be overloaded.
- Indexers can have more than one formal parameter, for example, when accessing a two-dimensional array.

Related Sections

- [Using Indexers](#)
- [Indexers in Interfaces](#)
- [Comparison Between Properties and Indexers](#)
- [Restricting Accessor Accessibility](#)

C# Language Specification

For more information, see [Indexers](#) in the [C# Language Specification](#). The language specification is the definitive source for C# syntax and usage.

See also

- [C# Programming Guide](#)
- [Properties](#)

Using indexers (C# Programming Guide)

12/28/2021 • 6 minutes to read • [Edit Online](#)

Indexers are a syntactic convenience that enable you to create a [class](#), [struct](#), or [interface](#) that client applications can access as an array. The compiler will generate an `Item` property (or an alternatively named property if `IndexerNameAttribute` is present), and the appropriate accessor methods. Indexers are most frequently implemented in types whose primary purpose is to encapsulate an internal collection or array. For example, suppose you have a class `TempRecord` that represents the temperature in Fahrenheit as recorded at 10 different times during a 24-hour period. The class contains a `temps` array of type `float[]` to store the temperature values. By implementing an indexer in this class, clients can access the temperatures in a `TempRecord` instance as `float temp = tempRecord[4]` instead of as `float temp = tempRecord.temps[4]`. The indexer notation not only simplifies the syntax for client applications; it also makes the class, and its purpose more intuitive for other developers to understand.

To declare an indexer on a class or struct, use the `this` keyword, as the following example shows:

```
// Indexer declaration
public int this[int index]
{
    // get and set accessors
}
```

IMPORTANT

Declaring an indexer will automatically generate a property named `Item` on the object. The `Item` property is not directly accessible from the instance [member access expression](#). Additionally, if you add your own `Item` property to an object with an indexer, you'll get a [CS0102 compiler error](#). To avoid this error, use the `IndexerNameAttribute` to rename the indexer as detailed below.

Remarks

The type of an indexer and the type of its parameters must be at least as accessible as the indexer itself. For more information about accessibility levels, see [Access Modifiers](#).

For more information about how to use indexers with an interface, see [Interface Indexers](#).

The signature of an indexer consists of the number and types of its formal parameters. It doesn't include the indexer type or the names of the formal parameters. If you declare more than one indexer in the same class, they must have different signatures.

An indexer value is not classified as a variable; therefore, you cannot pass an indexer value as a [ref](#) or [out](#) parameter.

To provide the indexer with a name that other languages can use, use `System.Runtime.CompilerServices.IndexerNameAttribute`, as the following example shows:


```
// Indexer declaration
[System.Runtime.CompilerServices.IndexerName("TheItem")]
public int this[int index]
{
    // get and set accessors
}
```

This indexer will have the name `TheItem`, as it is overridden by the indexer name attribute. By default, the indexer name is `Item`.

Example 1

The following example shows how to declare a private array field, `temps`, and an indexer. The indexer enables direct access to the instance `tempRecord[i]`. The alternative to using the indexer is to declare the array as a [public](#) member and access its members, `tempRecord.temps[i]`, directly.

```
public class TempRecord
{
    // Array of temperature values
    float[] temps = new float[10]
    {
        56.2F, 56.7F, 56.5F, 56.9F, 58.8F,
        61.3F, 65.9F, 62.1F, 59.2F, 57.5F
    };

    // To enable client code to validate input
    // when accessing your indexer.
    public int Length => temps.Length;

    // Indexer declaration.
    // If index is out of range, the temps array will throw the exception.
    public float this[int index]
    {
        get => temps[index];
        set => temps[index] = value;
    }
}
```

Notice that when an indexer's access is evaluated, for example, in a `Console.Write` statement, the [get](#) accessor is invoked. Therefore, if no `get` accessor exists, a compile-time error occurs.

```

using System;

class Program
{
    static void Main()
    {
        var tempRecord = new TempRecord();

        // Use the indexer's set accessor
        tempRecord[3] = 58.3F;
        tempRecord[5] = 60.1F;

        // Use the indexer's get accessor
        for (int i = 0; i < 10; i++)
        {
            Console.WriteLine($"Element #{i} = {tempRecord[i]}");
        }

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
    /* Output:
        Element #0 = 56.2
        Element #1 = 56.7
        Element #2 = 56.5
        Element #3 = 58.3
        Element #4 = 58.8
        Element #5 = 60.1
        Element #6 = 65.9
        Element #7 = 62.1
        Element #8 = 59.2
        Element #9 = 57.5
    */
}

```

Indexing using other values

C# doesn't limit the indexer parameter type to integer. For example, it may be useful to use a string with an indexer. Such an indexer might be implemented by searching for the string in the collection, and returning the appropriate value. As accessors can be overloaded, the string and integer versions can coexist.

Example 2

The following example declares a class that stores the days of the week. A `get` accessor takes a string, the name of a day, and returns the corresponding integer. For example, "Sunday" returns 0, "Monday" returns 1, and so on.

```

using System;

// Using a string as an indexer value
class DayCollection
{
    string[] days = { "Sun", "Mon", "Tues", "Wed", "Thurs", "Fri", "Sat" };

    // Indexer with only a get accessor with the expression-bodied definition:
    public int this[string day] => FindDayIndex(day);

    private int FindDayIndex(string day)
    {
        for (int j = 0; j < days.Length; j++)
        {
            if (days[j] == day)
            {
                return j;
            }
        }

        throw new ArgumentOutOfRangeException(
            nameof(day),
            $"Day {day} is not supported.\nDay input must be in the form \"Sun\", \"Mon\", etc");
    }
}

```

Consuming example 2

```

using System;

class Program
{
    static void Main(string[] args)
    {
        var week = new DayCollection();
        Console.WriteLine(week["Fri"]);

        try
        {
            Console.WriteLine(week["Made-up day"]);
        }
        catch (ArgumentOutOfRangeException e)
        {
            Console.WriteLine($"Not supported input: {e.Message}");
        }
    }
}
// Output:
// 5
// Not supported input: Day Made-up day is not supported.
// Day input must be in the form "Sun", "Mon", etc (Parameter 'day')

```

Example 3

The following example declares a class that stores the days of the week using the [System.DayOfWeek](#) enum. A `get` accessor takes a `DayOfWeek`, the value of a day, and returns the corresponding integer. For example, `DayOfWeek.Sunday` returns 0, `DayOfWeek.Monday` returns 1, and so on.

```

using System;
using Day = System.DayOfWeek;

class DayOfWeekCollection
{
    Day[] days =
    {
        Day.Sunday, Day.Monday, Day.Tuesday, Day.Wednesday,
        Day.Thursday, Day.Friday, Day.Saturday
    };

    // Indexer with only a get accessor with the expression-bodied definition:
    public int this[Day day] => FindDayIndex(day);

    private int FindDayIndex(Day day)
    {
        for (int j = 0; j < days.Length; j++)
        {
            if (days[j] == day)
            {
                return j;
            }
        }
        throw new ArgumentOutOfRangeException(
            nameof(day),
            $"Day {day} is not supported.\nDay input must be a defined System.DayOfWeek value.");
    }
}

```

Consuming example 3

```

using System;

class Program
{
    static void Main()
    {
        var week = new DayOfWeekCollection();
        Console.WriteLine(week[DayOfWeek.Friday]);

        try
        {
            Console.WriteLine(week[(DayOfWeek)43]);
        }
        catch (ArgumentOutOfRangeException e)
        {
            Console.WriteLine($"Not supported input: {e.Message}");
        }
    }
    // Output:
    // 5
    // Not supported input: Day 43 is not supported.
    // Day input must be a defined System.DayOfWeek value. (Parameter 'day')
}

```

Robust programming

There are two main ways in which the security and reliability of indexers can be improved:

- Be sure to incorporate some type of error-handling strategy to handle the chance of client code passing in an invalid index value. In the first example earlier in this topic, the TempRecord class provides a Length property that enables the client code to verify the input before passing it to the indexer. You can also put the error handling code inside the indexer itself. Be sure to document for users any exceptions that you

throw inside an indexer accessor.

- Set the accessibility of the [get](#) and [set](#) accessors to be as restrictive as is reasonable. This is important for the `set` accessor in particular. For more information, see [Restricting Accessor Accessibility](#).

See also

- [C# Programming Guide](#)
- [Indexers](#)
- [Properties](#)

Indexers in Interfaces (C# Programming Guide)

12/28/2021 • 2 minutes to read • [Edit Online](#)

Indexers can be declared on an [interface](#). Accessors of interface indexers differ from the accessors of [class](#) indexers in the following ways:

- Interface accessors do not use modifiers.
- An interface accessor typically does not have a body.

The purpose of the accessor is to indicate whether the indexer is read-write, read-only, or write-only. You may provide an implementation for an indexer defined in an interface, but this is rare. Indexers typically define an API to access data fields, and data fields cannot be defined in an interface.

The following is an example of an interface indexer accessor:

```
public interface ISomeInterface
{
    //...

    // Indexer declaration:
    string this[int index]
    {
        get;
        set;
    }
}
```

The signature of an indexer must differ from the signatures of all other indexers declared in the same interface.

Example

The following example shows how to implement interface indexers.

```
// Indexer on an interface:
public interface IIndexInterface
{
    // Indexer declaration:
    int this[int index]
    {
        get;
        set;
    }
}

// Implementing the interface.
class IndexerClass : IIndexInterface
{
    private int[] arr = new int[100];
    public int this[int index] // indexer declaration
    {
        // The arr object will throw IndexOutOfRangeException exception.
        get => arr[index];
        set => arr[index] = value;
    }
}
```

```

IndexerClass test = new IndexerClass();
System.Random rand = new System.Random();
// Call the indexer to initialize its elements.
for (int i = 0; i < 10; i++)
{
    test[i] = rand.Next();
}
for (int i = 0; i < 10; i++)
{
    System.Console.WriteLine($"Element #{i} = {test[i]}");
}

/* Sample output:
    Element #0 = 360877544
    Element #1 = 327058047
    Element #2 = 1913480832
    Element #3 = 1519039937
    Element #4 = 601472233
    Element #5 = 323352310
    Element #6 = 1422639981
    Element #7 = 1797892494
    Element #8 = 875761049
    Element #9 = 393083859
*/

```

In the preceding example, you could use the explicit interface member implementation by using the fully qualified name of the interface member. For example

```

string IIndexInterface.this[int index]
{
}

```

However, the fully qualified name is only needed to avoid ambiguity when the class is implementing more than one interface with the same indexer signature. For example, if an `Employee` class is implementing two interfaces, `ICitizen` and `IEmployee`, and both interfaces have the same indexer signature, the explicit interface member implementation is necessary. That is, the following indexer declaration:

```

string IEmployee.this[int index]
{
}

```

implements the indexer on the `IEmployee` interface, while the following declaration:

```

string ICitizen.this[int index]
{
}

```

implements the indexer on the `ICitizen` interface.

See also

- [C# Programming Guide](#)
- [Indexers](#)
- [Properties](#)
- [Interfaces](#)

Comparison Between Properties and Indexers (C# Programming Guide)

12/28/2021 • 2 minutes to read • [Edit Online](#)

Indexers are like properties. Except for the differences shown in the following table, all the rules that are defined for property accessors apply to indexer accessors also.

PROPERTY	INDEXER
Allows methods to be called as if they were public data members.	Allows elements of an internal collection of an object to be accessed by using array notation on the object itself.
Accessed through a simple name.	Accessed through an index.
Can be a static or an instance member.	Must be an instance member.
A get accessor of a property has no parameters.	A <code>get</code> accessor of an indexer has the same formal parameter list as the indexer.
A set accessor of a property contains the implicit <code>value</code> parameter.	A <code>set</code> accessor of an indexer has the same formal parameter list as the indexer, and also to the value parameter.
Supports shortened syntax with Auto-Implemented Properties .	Supports expression bodied members for get only indexers.

See also

- [C# Programming Guide](#)
- [Indexers](#)
- [Properties](#)

Events (C# Programming Guide)

12/28/2021 • 2 minutes to read • [Edit Online](#)

Events enable a [class](#) or object to notify other classes or objects when something of interest occurs. The class that sends (or *raises*) the event is called the *publisher* and the classes that receive (or *handle*) the event are called *subscribers*.

In a typical C# Windows Forms or Web application, you subscribe to events raised by controls such as buttons and list boxes. You can use the Visual C# integrated development environment (IDE) to browse the events that a control publishes and select the ones that you want to handle. The IDE provides an easy way to automatically add an empty event handler method and the code to subscribe to the event. For more information, see [How to subscribe to and unsubscribe from events](#).

Events Overview

Events have the following properties:

- The publisher determines when an event is raised; the subscribers determine what action is taken in response to the event.
- An event can have multiple subscribers. A subscriber can handle multiple events from multiple publishers.
- Events that have no subscribers are never raised.
- Events are typically used to signal user actions such as button clicks or menu selections in graphical user interfaces.
- When an event has multiple subscribers, the event handlers are invoked synchronously when an event is raised. To invoke events asynchronously, see [Calling Synchronous Methods Asynchronously](#).
- In the .NET class library, events are based on the [EventHandler](#) delegate and the [EventArgs](#) base class.

Related Sections

For more information, see:

- [How to subscribe to and unsubscribe from events](#)
- [How to publish events that conform to .NET Guidelines](#)
- [How to raise base class events in derived classes](#)
- [How to implement interface events](#)
- [How to implement custom event accessors](#)

C# Language Specification

For more information, see [Events](#) in the [C# Language Specification](#). The language specification is the definitive source for C# syntax and usage.

Featured Book Chapters

[Delegates, Events, and Lambda Expressions](#) in [C# 3.0 Cookbook, Third Edition: More than 250 solutions for C# 3.0 programmers](#)

[Delegates and Events](#) in [Learning C# 3.0: Fundamentals of C# 3.0](#)

See also

- [EventHandler](#)
- [C# Programming Guide](#)
- [Delegates](#)
- [Creating Event Handlers in Windows Forms](#)

How to subscribe to and unsubscribe from events (C# Programming Guide)

12/28/2021 • 3 minutes to read • [Edit Online](#)

You subscribe to an event that is published by another class when you want to write custom code that is called when that event is raised. For example, you might subscribe to a button's `click` event in order to make your application do something useful when the user clicks the button.

To subscribe to events by using the Visual Studio IDE

1. If you cannot see the **Properties** window, in **Design** view, right-click the form or control for which you want to create an event handler, and select **Properties**.
2. On top of the **Properties** window, click the **Events** icon.
3. Double-click the event that you want to create, for example the `Load` event.

Visual C# creates an empty event handler method and adds it to your code. Alternatively you can add the code manually in **Code** view. For example, the following lines of code declare an event handler method that will be called when the `Form` class raises the `Load` event.

```
private void Form1_Load(object sender, System.EventArgs e)
{
    // Add your form load event handling code here.
}
```

The line of code that is required to subscribe to the event is also automatically generated in the `InitializeComponent` method in the `Form1.Designer.cs` file in your project. It resembles this:

```
this.Load += new System.EventHandler(this.Form1_Load);
```

To subscribe to events programmatically

1. Define an event handler method whose signature matches the delegate signature for the event. For example, if the event is based on the `EventHandler` delegate type, the following code represents the method stub:

```
void HandleCustomEvent(object sender, CustomEventArgs a)
{
    // Do something useful here.
}
```

2. Use the addition assignment operator (`+=`) to attach an event handler to the event. In the following example, assume that an object named `publisher` has an event named `RaiseCustomEvent`. Note that the subscriber class needs a reference to the publisher class in order to subscribe to its events.

```
publisher.RaiseCustomEvent += HandleCustomEvent;
```

You can also use a [lambda expression](#) to specify an event handler:

```
public Form1()
{
    InitializeComponent();
    this.Click += (s,e) =>
    {
        MessageBox.Show(((MouseEventArgs)e).Location.ToString());
    };
}
```

To subscribe to events by using an anonymous function

If you don't have to unsubscribe from an event later, you can use the addition assignment operator (`+=`) to attach an anonymous function as an event handler. In the following example, assume that an object named `publisher` has an event named `RaiseCustomEvent` and that a `CustomEventArgs` class has also been defined to carry some kind of specialized event information. Note that the subscriber class needs a reference to `publisher` in order to subscribe to its events.

```
publisher.RaiseCustomEvent += (object o, CustomEventArgs e) =>
{
    string s = o.ToString() + " " + e.ToString();
    Console.WriteLine(s);
};
```

You cannot easily unsubscribe from an event if you used an anonymous function to subscribe to it. To unsubscribe in this scenario, go back to the code where you subscribe to the event, store the anonymous function in a delegate variable, and then add the delegate to the event. We recommend that you don't use anonymous functions to subscribe to events if you have to unsubscribe from the event at some later point in your code. For more information about anonymous functions, see [Lambda expressions](#).

Unsubscribing

To prevent your event handler from being invoked when the event is raised, unsubscribe from the event. In order to prevent resource leaks, you should unsubscribe from events before you dispose of a subscriber object. Until you unsubscribe from an event, the multicast delegate that underlies the event in the publishing object has a reference to the delegate that encapsulates the subscriber's event handler. As long as the publishing object holds that reference, garbage collection will not delete your subscriber object.

To unsubscribe from an event

- Use the subtraction assignment operator (`-=`) to unsubscribe from an event:

```
publisher.RaiseCustomEvent -= HandleCustomEvent;
```

When all subscribers have unsubscribed from an event, the event instance in the publisher class is set to `null`.

See also

- [Events](#)
- [event](#)
- [How to publish events that conform to .NET Guidelines](#)
- [- and -= operators](#)
- [+ and += operators](#)

How to publish events that conform to .NET Guidelines (C# Programming Guide)

12/28/2021 • 3 minutes to read • [Edit Online](#)

The following procedure demonstrates how to add events that follow the standard .NET pattern to your classes and structs. All events in the .NET class library are based on the [EventHandler](#) delegate, which is defined as follows:

```
public delegate void EventHandler(object sender, EventArgs e);
```

NOTE

.NET Framework 2.0 introduces a generic version of this delegate, [EventHandler<TEventArgs>](#). The following examples show how to use both versions.

Although events in classes that you define can be based on any valid delegate type, even delegates that return a value, it is generally recommended that you base your events on the .NET pattern by using [EventHandler](#), as shown in the following example.

The name `EventHandler` can lead to a bit of confusion as it doesn't actually handle the event. The [EventHandler](#), and generic [EventHandler<TEventArgs>](#) are delegate types. A method or lambda expression whose signature matches the delegate definition is the *event handler* and will be invoked when the event is raised.

Publish events based on the EventHandler pattern

1. (Skip this step and go to Step 3a if you do not have to send custom data with your event.) Declare the class for your custom data at a scope that is visible to both your publisher and subscriber classes. Then add the required members to hold your custom event data. In this example, a simple string is returned.

```
public class CustomEventArgs : EventArgs
{
    public CustomEventArgs(string message)
    {
        Message = message;
    }

    public string Message { get; set; }
}
```

2. (Skip this step if you are using the generic version of [EventHandler<TEventArgs>](#).) Declare a delegate in your publishing class. Give it a name that ends with `EventHandler`. The second parameter specifies your custom `EventArgs` type.

```
public delegate void CustomEventHandler(object sender, CustomEventArgs args);
```

3. Declare the event in your publishing class by using one of the following steps.
 - a. If you have no custom EventArgs class, your Event type will be the non-generic EventHandler delegate. You do not have to declare the delegate because it is already declared in the [System](#)

namespace that is included when you create your C# project. Add the following code to your publisher class.

```
public event EventHandler RaiseCustomEvent;
```

- b. If you are using the non-generic version of [EventHandler](#) and you have a custom class derived from [EventArgs](#), declare your event inside your publishing class and use your delegate from step 2 as the type.

```
public event CustomEventHandler RaiseCustomEvent;
```

- c. If you are using the generic version, you do not need a custom delegate. Instead, in your publishing class, you specify your event type as `EventHandler<CustomEventArgs>`, substituting the name of your own class between the angle brackets.

```
public event EventHandler<CustomEventArgs> RaiseCustomEvent;
```

Example

The following example demonstrates the previous steps by using a custom `EventArgs` class and `EventHandler<TEventArgs>` as the event type.

```
using System;

namespace DotNetEvents
{
    // Define a class to hold custom event info
    public class CustomEventArgs : EventArgs
    {
        public CustomEventArgs(string message)
        {
            Message = message;
        }

        public string Message { get; set; }
    }

    // Class that publishes an event
    class Publisher
    {
        // Declare the event using EventHandler<T>
        public event EventHandler<CustomEventArgs> RaiseCustomEvent;

        public void DoSomething()
        {
            // Write some code that does something useful here
            // then raise the event. You can also raise an event
            // before you execute a block of code.
            OnRaiseCustomEvent(new CustomEventArgs("Event triggered"));
        }

        // Wrap event invocations inside a protected virtual method
        // to allow derived classes to override the event invocation behavior
        protected virtual void OnRaiseCustomEvent(CustomEventArgs e)
        {
            // Make a temporary copy of the event to avoid possibility of
            // a race condition if the last subscriber unsubscribes
            // immediately after the null check and before the event is raised.
            EventHandler<CustomEventArgs> raiseEvent = RaiseCustomEvent;
```

```

        // Event will be null if there are no subscribers
        if (raiseEvent != null)
        {
            // Format the string to send inside the CustomEventArgs parameter
            e.Message += $" at {DateTime.Now}";

            // Call to raise the event.
            raiseEvent(this, e);
        }
    }
}

//Class that subscribes to an event
class Subscriber
{
    private readonly string _id;

    public Subscriber(string id, Publisher pub)
    {
        _id = id;

        // Subscribe to the event
        pub.RaiseCustomEvent += HandleCustomEvent;
    }

    // Define what actions to take when the event is raised.
    void HandleCustomEvent(object sender, CustomEventArgs e)
    {
        Console.WriteLine($"{_id} received this message: {e.Message}");
    }
}

class Program
{
    static void Main()
    {
        var pub = new Publisher();
        var sub1 = new Subscriber("sub1", pub);
        var sub2 = new Subscriber("sub2", pub);

        // Call the method that raises the event.
        pub.DoSomething();

        // Keep the console window open
        Console.WriteLine("Press any key to continue...");
        Console.ReadLine();
    }
}

```

See also

- [Delegate](#)
- [C# Programming Guide](#)
- [Events](#)
- [Delegates](#)

How to raise base class events in derived classes (C# Programming Guide)

12/28/2021 • 3 minutes to read • [Edit Online](#)

The following simple example shows the standard way to declare events in a base class so that they can also be raised from derived classes. This pattern is used extensively in Windows Forms classes in the .NET class libraries.

When you create a class that can be used as a base class for other classes, you should consider the fact that events are a special type of delegate that can only be invoked from within the class that declared them. Derived classes cannot directly invoke events that are declared within the base class. Although sometimes you may want an event that can only be raised by the base class, most of the time, you should enable the derived class to invoke base class events. To do this, you can create a protected invoking method in the base class that wraps the event. By calling or overriding this invoking method, derived classes can invoke the event indirectly.

NOTE

Do not declare virtual events in a base class and override them in a derived class. The C# compiler does not handle these correctly and it is unpredictable whether a subscriber to the derived event will actually be subscribing to the base class event.

Example

```
namespace BaseClassEvents
{
    // Special EventArgs class to hold info about Shapes.
    public class ShapeEventArgs : EventArgs
    {
        public ShapeEventArgs(double area)
        {
            NewArea = area;
        }

        public double NewArea { get; }
    }

    // Base class event publisher
    public abstract class Shape
    {
        protected double _area;

        public double Area
        {
            get => _area;
            set => _area = value;
        }

        // The event. Note that by using the generic EventHandler<T> event type
        // we do not need to declare a separate delegate type.
        public event EventHandler<ShapeEventArgs> ShapeChanged;

        public abstract void Draw();

        //The event-invoking method that derived classes can override.
        protected virtual void OnShapeChanged(ShapeEventArgs e)
        {
        }
    }
}
```



```

        // Safely raise the event for all subscribers
        ShapeChanged?.Invoke(this, e);
    }
}

public class Circle : Shape
{
    private double _radius;

    public Circle(double radius)
    {
        _radius = radius;
        _area = 3.14 * _radius * _radius;
    }

    public void Update(double d)
    {
        _radius = d;
        _area = 3.14 * _radius * _radius;
        OnShapeChanged(new ShapeEventArgs(_area));
    }

    protected override void OnShapeChanged(ShapeEventArgs e)
    {
        // Do any circle-specific processing here.

        // Call the base class event invocation method.
        base.OnShapeChanged(e);
    }

    public override void Draw()
    {
        Console.WriteLine("Drawing a circle");
    }
}

public class Rectangle : Shape
{
    private double _length;
    private double _width;

    public Rectangle(double length, double width)
    {
        _length = length;
        _width = width;
        _area = _length * _width;
    }

    public void Update(double length, double width)
    {
        _length = length;
        _width = width;
        _area = _length * _width;
        OnShapeChanged(new ShapeEventArgs(_area));
    }

    protected override void OnShapeChanged(ShapeEventArgs e)
    {
        // Do any rectangle-specific processing here.

        // Call the base class event invocation method.
        base.OnShapeChanged(e);
    }

    public override void Draw()
    {
        Console.WriteLine("Drawing a rectangle");
    }
}

```

```

    }

    // Represents the surface on which the shapes are drawn
    // Subscribes to shape events so that it knows
    // when to redraw a shape.
    public class ShapeContainer
    {
        private readonly List<Shape> _list;

        public ShapeContainer()
        {
            _list = new List<Shape>();
        }

        public void AddShape(Shape shape)
        {
            _list.Add(shape);

            // Subscribe to the base class event.
            shape.ShapeChanged += HandleShapeChanged;
        }

        // ...Other methods to draw, resize, etc.

        private void HandleShapeChanged(object sender, ShapeEventArgs e)
        {
            if (sender is Shape shape)
            {
                // Diagnostic message for demonstration purposes.
                Console.WriteLine($"Received event. Shape area is now {e.NewArea}");

                // Redraw the shape here.
                shape.Draw();
            }
        }
    }

    class Test
    {
        static void Main()
        {
            //Create the event publishers and subscriber
            var circle = new Circle(54);
            var rectangle = new Rectangle(12, 9);
            var container = new ShapeContainer();

            // Add the shapes to the container.
            container.AddShape(circle);
            container.AddShape(rectangle);

            // Cause some events to be raised.
            circle.Update(57);
            rectangle.Update(7, 7);

            // Keep the console window open in debug mode.
            Console.WriteLine("Press any key to continue...");
            Console.ReadKey();
        }
    }
}

/* Output:
    Received event. Shape area is now 10201.86
    Drawing a circle
    Received event. Shape area is now 49
    Drawing a rectangle
*/

```

See also

- [C# Programming Guide](#)
- [Events](#)
- [Delegates](#)
- [Access Modifiers](#)
- [Creating Event Handlers in Windows Forms](#)

How to implement interface events (C# Programming Guide)

12/28/2021 • 3 minutes to read • [Edit Online](#)

An [interface](#) can declare an [event](#). The following example shows how to implement interface events in a class. Basically the rules are the same as when you implement any interface method or property.

To implement interface events in a class

Declare the event in your class and then invoke it in the appropriate areas.

```
namespace ImplementInterfaceEvents
{
    public interface IDrawingObject
    {
        event EventHandler ShapeChanged;
    }
    public class MyEventArgs : EventArgs
    {
        // class members
    }
    public class Shape : IDrawingObject
    {
        public event EventHandler ShapeChanged;
        void ChangeShape()
        {
            // Do something here before the event...

            OnShapeChanged(new MyEventArgs(/*arguments*/));

            // or do something here after the event.
        }
        protected virtual void OnShapeChanged(MyEventArgs e)
        {
            ShapeChanged?.Invoke(this, e);
        }
    }
}
```

Example

The following example shows how to handle the less-common situation in which your class inherits from two or more interfaces and each interface has an event with the same name. In this situation, you must provide an explicit interface implementation for at least one of the events. When you write an explicit interface implementation for an event, you must also write the `add` and `remove` event accessors. Normally these are provided by the compiler, but in this case the compiler cannot provide them.

By providing your own accessors, you can specify whether the two events are represented by the same event in your class, or by different events. For example, if the events should be raised at different times according to the interface specifications, you can associate each event with a separate implementation in your class. In the following example, subscribers determine which `OnDraw` event they will receive by casting the shape reference to either an `IShape` or an `IDrawingObject`.

```

namespace WrapTwoInterfaceEvents
{
    using System;

    public interface IDrawingObject
    {
        // Raise this event before drawing
        // the object.
        event EventHandler OnDraw;
    }

    public interface IShape
    {
        // Raise this event after drawing
        // the shape.
        event EventHandler OnDraw;
    }

    // Base class event publisher inherits two
    // interfaces, each with an OnDraw event
    public class Shape : IDrawingObject, IShape
    {
        // Create an event for each interface event
        event EventHandler PreDrawEvent;
        event EventHandler PostDrawEvent;

        object objectLock = new Object();

        // Explicit interface implementation required.
        // Associate IDrawingObject's event with
        // PreDrawEvent
        #region IDrawingObjectOnDraw
        event EventHandler IDrawingObject.OnDraw
        {
            add
            {
                lock (objectLock)
                {
                    PreDrawEvent += value;
                }
            }
            remove
            {
                lock (objectLock)
                {
                    PreDrawEvent -= value;
                }
            }
        }
        #endregion
        // Explicit interface implementation required.
        // Associate IShape's event with
        // PostDrawEvent
        event EventHandler IShape.OnDraw
        {
            add
            {
                lock (objectLock)
                {
                    PostDrawEvent += value;
                }
            }
            remove
            {
                lock (objectLock)
                {
                    PostDrawEvent -= value;
                }
            }
        }
    }
}

```

```

// For the sake of simplicity this one method
// implements both interfaces.
public void Draw()
{
    // Raise IDrawingObject's event before the object is drawn.
    PreDrawEvent?.Invoke(this, EventArgs.Empty);

    Console.WriteLine("Drawing a shape.");

    // Raise IShape's event after the object is drawn.
    PostDrawEvent?.Invoke(this, EventArgs.Empty);
}
}
public class Subscriber1
{
    // References the shape object as an IDrawingObject
    public Subscriber1(Shape shape)
    {
        IDrawingObject d = (IDrawingObject)shape;
        d.OnDraw += d_OnDraw;
    }

    void d_OnDraw(object sender, EventArgs e)
    {
        Console.WriteLine("Sub1 receives the IDrawingObject event.");
    }
}
// References the shape object as an IShape
public class Subscriber2
{
    public Subscriber2(Shape shape)
    {
        IShape d = (IShape)shape;
        d.OnDraw += d_OnDraw;
    }

    void d_OnDraw(object sender, EventArgs e)
    {
        Console.WriteLine("Sub2 receives the IShape event.");
    }
}

public class Program
{
    static void Main(string[] args)
    {
        Shape shape = new Shape();
        Subscriber1 sub = new Subscriber1(shape);
        Subscriber2 sub2 = new Subscriber2(shape);
        shape.Draw();

        // Keep the console window open in debug mode.
        System.Console.WriteLine("Press any key to exit.");
        System.Console.ReadKey();
    }
}
}
/* Output:
Sub1 receives the IDrawingObject event.
Drawing a shape.
Sub2 receives the IShape event.
*/

```

See also

- [C# Programming Guide](#)
- [Events](#)
- [Delegates](#)
- [Explicit Interface Implementation](#)
- [How to raise base class events in derived classes](#)

How to implement custom event accessors (C# Programming Guide)

12/28/2021 • 2 minutes to read • [Edit Online](#)

An event is a special kind of multicast delegate that can only be invoked from within the class that it is declared in. Client code subscribes to the event by providing a reference to a method that should be invoked when the event is fired. These methods are added to the delegate's invocation list through event accessors, which resemble property accessors, except that event accessors are named `add` and `remove`. In most cases, you do not have to supply custom event accessors. When no custom event accessors are supplied in your code, the compiler will add them automatically. However, in some cases you may have to provide custom behavior. One such case is shown in the topic [How to implement interface events](#).

Example

The following example shows how to implement custom add and remove event accessors. Although you can substitute any code inside the accessors, we recommend that you lock the event before you add or remove a new event handler method.

```
event EventHandler IDrawingObject.OnDraw
{
    add
    {
        lock (objectLock)
        {
            PreDrawEvent += value;
        }
    }
    remove
    {
        lock (objectLock)
        {
            PreDrawEvent -= value;
        }
    }
}
```

See also

- [Events](#)
- [event](#)

Generic type parameters (C# Programming Guide)

12/28/2021 • 2 minutes to read • [Edit Online](#)

In a generic type or method definition, a type parameter is a placeholder for a specific type that a client specifies when they create an instance of the generic type. A generic class, such as `GenericList<T>` listed in [Introduction to Generics](#), cannot be used as-is because it is not really a type; it is more like a blueprint for a type. To use `GenericList<T>`, client code must declare and instantiate a constructed type by specifying a type argument inside the angle brackets. The type argument for this particular class can be any type recognized by the compiler. Any number of constructed type instances can be created, each one using a different type argument, as follows:

```
GenericList<float> list1 = new GenericList<float>();
GenericList<ExampleClass> list2 = new GenericList<ExampleClass>();
GenericList<ExampleStruct> list3 = new GenericList<ExampleStruct>();
```

In each of these instances of `GenericList<T>`, every occurrence of `T` in the class is substituted at run time with the type argument. By means of this substitution, we have created three separate type-safe and efficient objects using a single class definition. For more information on how this substitution is performed by the CLR, see [Generics in the Run Time](#).

Type parameter naming guidelines

- **Do** name generic type parameters with descriptive names, unless a single letter name is completely self explanatory and a descriptive name would not add value.

```
public interface ISessionChannel<TSession> { /*...*/ }
public delegate TOutput Converter<TInput, TOutput>(TInput from);
public class List<T> { /*...*/ }
```

- **Consider** using `T` as the type parameter name for types with one single letter type parameter.

```
public int IComparer<T>() { return 0; }
public delegate bool Predicate<T>(T item);
public struct Nullable<T> where T : struct { /*...*/ }
```

- **Do** prefix descriptive type parameter names with "T".

```
public interface ISessionChannel<TSession>
{
    TSession Session { get; }
}
```

- **Consider** indicating constraints placed on a type parameter in the name of parameter. For example, a parameter constrained to `ISession` may be called `TSession`.

The code analysis rule [CA1715](#) can be used to ensure that type parameters are named appropriately.

See also

- [System.Collections.Generic](#)

- [C# Programming Guide](#)
- [Generics](#)
- [Differences Between C++ Templates and C# Generics](#)

Constraints on type parameters (C# Programming Guide)

12/28/2021 • 11 minutes to read • [Edit Online](#)

Constraints inform the compiler about the capabilities a type argument must have. Without any constraints, the type argument could be any type. The compiler can only assume the members of [System.Object](#), which is the ultimate base class for any .NET type. For more information, see [Why use constraints](#). If client code uses a type that doesn't satisfy a constraint, the compiler issues an error. Constraints are specified by using the `where` contextual keyword. The following table lists the various types of constraints:

CONSTRAINT	DESCRIPTION
<code>where T : struct</code>	The type argument must be a non-nullable value type . For information about nullable value types, see Nullable value types . Because all value types have an accessible parameterless constructor, the <code>struct</code> constraint implies the <code>new()</code> constraint and can't be combined with the <code>new()</code> constraint. You can't combine the <code>struct</code> constraint with the <code>unmanaged</code> constraint.
<code>where T : class</code>	The type argument must be a reference type. This constraint applies also to any class, interface, delegate, or array type. In a nullable context in C# 8.0 or later, <code>T</code> must be a non-nullable reference type.
<code>where T : class?</code>	The type argument must be a reference type, either nullable or non-nullable. This constraint applies also to any class, interface, delegate, or array type.
<code>where T : notnull</code>	The type argument must be a non-nullable type. The argument can be a non-nullable reference type in C# 8.0 or later, or a non-nullable value type.
<code>where T : default</code>	This constraint resolves the ambiguity when you need to specify an unconstrained type parameter when you override a method or provide an explicit interface implementation. The <code>default</code> constraint implies the base method without either the <code>class</code> or <code>struct</code> constraint. For more information, see the default constraint spec proposal.
<code>where T : unmanaged</code>	The type argument must be a non-nullable unmanaged type . The <code>unmanaged</code> constraint implies the <code>struct</code> constraint and can't be combined with either the <code>struct</code> or <code>new()</code> constraints.
<code>where T : new()</code>	The type argument must have a public parameterless constructor. When used together with other constraints, the <code>new()</code> constraint must be specified last. The <code>new()</code> constraint can't be combined with the <code>struct</code> and <code>unmanaged</code> constraints.

CONSTRAINT	DESCRIPTION
<code>where T : <base class name></code>	The type argument must be or derive from the specified base class. In a nullable context in C# 8.0 and later, <code>T</code> must be a non-nullable reference type derived from the specified base class.
<code>where T : <base class name>?</code>	The type argument must be or derive from the specified base class. In a nullable context in C# 8.0 and later, <code>T</code> may be either a nullable or non-nullable type derived from the specified base class.
<code>where T : <interface name></code>	The type argument must be or implement the specified interface. Multiple interface constraints can be specified. The constraining interface can also be generic. In a nullable context in C# 8.0 and later, <code>T</code> must be a non-nullable type that implements the specified interface.
<code>where T : <interface name>?</code>	The type argument must be or implement the specified interface. Multiple interface constraints can be specified. The constraining interface can also be generic. In a nullable context in C# 8.0, <code>T</code> may be a nullable reference type, a non-nullable reference type, or a value type. <code>T</code> may not be a nullable value type.
<code>where T : U</code>	The type argument supplied for <code>T</code> must be or derive from the argument supplied for <code>U</code> . In a nullable context, if <code>U</code> is a non-nullable reference type, <code>T</code> must be non-nullable reference type. If <code>U</code> is a nullable reference type, <code>T</code> may be either nullable or non-nullable.

Why use constraints

Constraints specify the capabilities and expectations of a type parameter. Declaring those constraints means you can use the operations and method calls of the constraining type. If your generic class or method uses any operation on the generic members beyond simple assignment or calling any methods not supported by [System.Object](#), you'll apply constraints to the type parameter. For example, the base class constraint tells the compiler that only objects of this type or derived from this type will be used as type arguments. Once the compiler has this guarantee, it can allow methods of that type to be called in the generic class. The following code example demonstrates the functionality you can add to the `GenericList<T>` class (in [Introduction to Generics](#)) by applying a base class constraint.

```

public class Employee
{
    public Employee(string name, int id) => (Name, ID) = (name, id);
    public string Name { get; set; }
    public int ID { get; set; }
}

public class GenericList<T> where T : Employee
{
    private class Node
    {
        public Node(T t) => (Next, Data) = (null, t);

        public Node Next { get; set; }
        public T Data { get; set; }
    }

    private Node head;

    public void AddHead(T t)
    {
        Node n = new Node(t) { Next = head };
        head = n;
    }

    public IEnumerator<T> GetEnumerator()
    {
        Node current = head;

        while (current != null)
        {
            yield return current.Data;
            current = current.Next;
        }
    }

    public T FindFirstOccurrence(string s)
    {
        Node current = head;
        T t = null;

        while (current != null)
        {
            //The constraint enables access to the Name property.
            if (current.Data.Name == s)
            {
                t = current.Data;
                break;
            }
            else
            {
                current = current.Next;
            }
        }
        return t;
    }
}

```

The constraint enables the generic class to use the `Employee.Name` property. The constraint specifies that all items of type `T` are guaranteed to be either an `Employee` object or an object that inherits from `Employee`.

Multiple constraints can be applied to the same type parameter, and the constraints themselves can be generic types, as follows:

```
class EmployeeList<T> where T : Employee, IEmployee, System.IComparable<T>, new()
{
    // ...
}
```

When applying the `where T : class` constraint, avoid the `==` and `!=` operators on the type parameter because these operators will test for reference identity only, not for value equality. This behavior occurs even if these operators are overloaded in a type that is used as an argument. The following code illustrates this point; the output is false even though the `String` class overloads the `==` operator.

```
public static void OpEqualsTest<T>(T s, T t) where T : class
{
    System.Console.WriteLine(s == t);
}

private static void TestStringEquality()
{
    string s1 = "target";
    System.Text.StringBuilder sb = new System.Text.StringBuilder("target");
    string s2 = sb.ToString();
    OpEqualsTest<string>(s1, s2);
}
```

The compiler only knows that `T` is a reference type at compile time and must use the default operators that are valid for all reference types. If you must test for value equality, the recommended way is to also apply the `where T : IEquatable<T>` or `where T : IComparable<T>` constraint and implement the interface in any class that will be used to construct the generic class.

Constraining multiple parameters

You can apply constraints to multiple parameters, and multiple constraints to a single parameter, as shown in the following example:

```
class Base { }
class Test<T, U>
    where U : struct
    where T : Base, new()
{ }
```

Unbounded type parameters

Type parameters that have no constraints, such as `T` in public class `SampleClass<T>{}`, are called unbounded type parameters. Unbounded type parameters have the following rules:

- The `!=` and `==` operators can't be used because there's no guarantee that the concrete type argument will support these operators.
- They can be converted to and from `System.Object` or explicitly converted to any interface type.
- You can compare them to `null`. If an unbounded parameter is compared to `null`, the comparison will always return false if the type argument is a value type.

Type parameters as constraints

The use of a generic type parameter as a constraint is useful when a member function with its own type parameter has to constrain that parameter to the type parameter of the containing type, as shown in the following example:

```
public class List<T>
{
    public void Add<U>(List<U> items) where U : T { /*...*/ }
}
```

In the previous example, `T` is a type constraint in the context of the `Add` method, and an unbounded type parameter in the context of the `List` class.

Type parameters can also be used as constraints in generic class definitions. The type parameter must be declared within the angle brackets together with any other type parameters:

```
//Type parameter V is used as a type constraint.
public class SampleClass<T, U, V> where T : V { }
```

The usefulness of type parameters as constraints with generic classes is limited because the compiler can assume nothing about the type parameter except that it derives from `System.Object`. Use type parameters as constraints on generic classes in scenarios in which you want to enforce an inheritance relationship between two type parameters.

nonnull constraint

Beginning with C# 8.0, you can use the `nonnull` constraint to specify that the type argument must be a non-nullable value type or non-nullable reference type. Unlike most other constraints, if a type argument violates the `nonnull` constraint, the compiler generates a warning instead of an error.

The `nonnull` constraint has an effect only when used in a nullable context. If you add the `nonnull` constraint in a nullable oblivious context, the compiler doesn't generate any warnings or errors for violations of the constraint.

class constraint

Beginning with C# 8.0, the `class` constraint in a nullable context specifies that the type argument must be a non-nullable reference type. In a nullable context, when a type argument is a nullable reference type, the compiler generates a warning.

default constraint

The addition of nullable reference types complicates the use of `T?` in a generic type or method. Prior to C# 8, `T?` could only be used when the `struct` constraint applied to `T`. In that context, `T?` refers to the `Nullable<T>` type for `T`. Starting with C# 8, `T?` could be used with either the `struct` or `class` constraint, but one of them must be present. When the `class` constraint was used, `T?` referred to the nullable reference type for `T`. Beginning with C# 9, `T?` can be used when neither constraint is applied. In that case, `T?` is interpreted the same as in C# 8 for value types and reference types. However, if `T` is an instance of `Nullable<T>`, `T?` is the same as `T`. In other words, it doesn't become `T??`.

Because `T?` can now be used without either the `class` or `struct` constraint, ambiguities can arise in overrides or explicit interface implementations. In both those cases, the override doesn't include the constraints, but inherits them from the base class. When the base class doesn't apply either the `class` or `struct` constraint, derived classes need to somehow specify an override applies to the base method without either constraint. That's when the derived method applies the `default` constraint. The `default` constraint clarifies *neither* the `class` nor `struct` constraint.

Unmanaged constraint

Beginning with C# 7.3, you can use the `unmanaged` constraint to specify that the type parameter must be a non-nullable [unmanaged type](#). The `unmanaged` constraint enables you to write reusable routines to work with types that can be manipulated as blocks of memory, as shown in the following example:

```
unsafe public static byte[] ToByteArray<T>(this T argument) where T : unmanaged
{
    var size = sizeof(T);
    var result = new Byte[size];
    Byte* p = (byte*)&argument;
    for (var i = 0; i < size; i++)
        result[i] = *p++;
    return result;
}
```

The preceding method must be compiled in an `unsafe` context because it uses the `sizeof` operator on a type not known to be a built-in type. Without the `unmanaged` constraint, the `sizeof` operator is unavailable.

The `unmanaged` constraint implies the `struct` constraint and can't be combined with it. Because the `struct` constraint implies the `new()` constraint, the `unmanaged` constraint can't be combined with the `new()` constraint as well.

Delegate constraints

Also beginning with C# 7.3, you can use [System.Delegate](#) or [System.MulticastDelegate](#) as a base class constraint. The CLR always allowed this constraint, but the C# language disallowed it. The `System.Delegate` constraint enables you to write code that works with delegates in a type-safe manner. The following code defines an extension method that combines two delegates provided they're the same type:

```
public static TDelegate TypeSafeCombine<TDelegate>(this TDelegate source, TDelegate target)
    where TDelegate : System.Delegate
    => Delegate.Combine(source, target) as TDelegate;
```

You can use the above method to combine delegates that are the same type:

```
Action first = () => Console.WriteLine("this");
Action second = () => Console.WriteLine("that");

var combined = first.TypeSafeCombine(second);
combined();

Func<bool> test = () => true;
// Combine signature ensures combined delegates must
// have the same type.
//var badCombined = first.TypeSafeCombine(test);
```

If you uncomment the last line, it won't compile. Both `first` and `test` are delegate types, but they're different delegate types.

Enum constraints

Beginning in C# 7.3, you can also specify the [System.Enum](#) type as a base class constraint. The CLR always allowed this constraint, but the C# language disallowed it. Generics using `System.Enum` provide type-safe programming to cache results from using the static methods in `System.Enum`. The following sample finds all the valid values for an enum type, and then builds a dictionary that maps those values to its string representation.


```
public static Dictionary<int, string> EnumNamedValues<T>() where T : System.Enum
{
    var result = new Dictionary<int, string>();
    var values = Enum.GetValues(typeof(T));

    foreach (int item in values)
        result.Add(item, Enum.GetName(typeof(T), item));
    return result;
}
```

`Enum.GetValues` and `Enum.GetName` use reflection, which has performance implications. You can call `EnumNamedValues` to build a collection that is cached and reused rather than repeating the calls that require reflection.

You could use it as shown in the following sample to create an enum and build a dictionary of its values and names:

```
enum Rainbow
{
    Red,
    Orange,
    Yellow,
    Green,
    Blue,
    Indigo,
    Violet
}
```

```
var map = EnumNamedValues<Rainbow>();

foreach (var pair in map)
    Console.WriteLine($"{pair.Key}: \t{pair.Value}");
```

See also

- [System.Collections.Generic](#)
- [C# Programming Guide](#)
- [Introduction to Generics](#)
- [Generic Classes](#)
- [new Constraint](#)

Generic Classes (C# Programming Guide)

12/28/2021 • 3 minutes to read • [Edit Online](#)

Generic classes encapsulate operations that are not specific to a particular data type. The most common use for generic classes is with collections like linked lists, hash tables, stacks, queues, trees, and so on. Operations such as adding and removing items from the collection are performed in basically the same way regardless of the type of data being stored.

For most scenarios that require collection classes, the recommended approach is to use the ones provided in the .NET class library. For more information about using these classes, see [Generic Collections in .NET](#).

Typically, you create generic classes by starting with an existing concrete class, and changing types into type parameters one at a time until you reach the optimal balance of generalization and usability. When creating your own generic classes, important considerations include the following:

- Which types to generalize into type parameters.

As a rule, the more types you can parameterize, the more flexible and reusable your code becomes. However, too much generalization can create code that is difficult for other developers to read or understand.

- What constraints, if any, to apply to the type parameters (See [Constraints on Type Parameters](#)).

A good rule is to apply the maximum constraints possible that will still let you handle the types you must handle. For example, if you know that your generic class is intended for use only with reference types, apply the class constraint. That will prevent unintended use of your class with value types, and will enable you to use the `as` operator on `T`, and check for null values.

- Whether to factor generic behavior into base classes and subclasses.

Because generic classes can serve as base classes, the same design considerations apply here as with non-generic classes. See the rules about inheriting from generic base classes later in this topic.

- Whether to implement one or more generic interfaces.

For example, if you are designing a class that will be used to create items in a generics-based collection, you may have to implement an interface such as `Comparable<T>` where `T` is the type of your class.

For an example of a simple generic class, see [Introduction to Generics](#).

The rules for type parameters and constraints have several implications for generic class behavior, especially regarding inheritance and member accessibility. Before proceeding, you should understand some terms. For a generic class `Node<T>`, client code can reference the class either by specifying a type argument, to create a closed constructed type (`Node<int>`). Alternatively, it can leave the type parameter unspecified, for example when you specify a generic base class, to create an open constructed type (`Node<T>`). Generic classes can inherit from concrete, closed constructed, or open constructed base classes:

```

class BaseNode { }
class BaseNodeGeneric<T> { }

// concrete type
class NodeConcrete<T> : BaseNode { }

//closed constructed type
class NodeClosed<T> : BaseNodeGeneric<int> { }

//open constructed type
class NodeOpen<T> : BaseNodeGeneric<T> { }

```

Non-generic, in other words, concrete, classes can inherit from closed constructed base classes, but not from open constructed classes or from type parameters because there is no way at run time for client code to supply the type argument required to instantiate the base class.

```

//No error
class Node1 : BaseNodeGeneric<int> { }

//Generates an error
//class Node2 : BaseNodeGeneric<T> {}

//Generates an error
//class Node3 : T {}

```

Generic classes that inherit from open constructed types must supply type arguments for any base class type parameters that are not shared by the inheriting class, as demonstrated in the following code:

```

class BaseNodeMultiple<T, U> { }

//No error
class Node4<T> : BaseNodeMultiple<T, int> { }

//No error
class Node5<T, U> : BaseNodeMultiple<T, U> { }

//Generates an error
//class Node6<T> : BaseNodeMultiple<T, U> {}

```

Generic classes that inherit from open constructed types must specify constraints that are a superset of, or imply, the constraints on the base type:

```

class NodeItem<T> where T : System.IComparable<T>, new() { }
class SpecialNodeItem<T> : NodeItem<T> where T : System.IComparable<T>, new() { }

```

Generic types can use multiple type parameters and constraints, as follows:

```

class SuperKeyType<K, V, U>
    where U : System.IComparable<U>
    where V : new()
{ }

```

Open constructed and closed constructed types can be used as method parameters:

```
void Swap<T>(List<T> list1, List<T> list2)
{
    //code to swap items
}

void Swap(List<int> list1, List<int> list2)
{
    //code to swap items
}
```

If a generic class implements an interface, all instances of that class can be cast to that interface.

Generic classes are invariant. In other words, if an input parameter specifies a `List<BaseClass>`, you will get a compile-time error if you try to provide a `List<DerivedClass>`.

See also

- [System.Collections.Generic](#)
- [C# Programming Guide](#)
- [Generics](#)
- [Saving the State of Enumerators](#)
- [An Inheritance Puzzle, Part One](#)

Generic Interfaces (C# Programming Guide)

12/28/2021 • 4 minutes to read • [Edit Online](#)

It is often useful to define interfaces either for generic collection classes, or for the generic classes that represent items in the collection. The preference for generic classes is to use generic interfaces, such as [IComparable<T>](#) rather than [IComparable](#), in order to avoid boxing and unboxing operations on value types. The .NET class library defines several generic interfaces for use with the collection classes in the [System.Collections.Generic](#) namespace.

When an interface is specified as a constraint on a type parameter, only types that implement the interface can be used. The following code example shows a `SortedList<T>` class that derives from the `GenericList<T>` class. For more information, see [Introduction to Generics](#). `SortedList<T>` adds the constraint `where T : IComparable<T>`. This enables the `BubbleSort` method in `SortedList<T>` to use the generic [CompareTo](#) method on list elements. In this example, list elements are a simple class, `Person`, that implements `IComparable<Person>`.

```
//Type parameter T in angle brackets.
public class GenericList<T> : System.Collections.Generic.IEnumerable<T>
{
    protected Node head;
    protected Node current = null;

    // Nested class is also generic on T
    protected class Node
    {
        public Node next;
        private T data; //T as private member datatype

        public Node(T t) //T used in non-generic constructor
        {
            next = null;
            data = t;
        }

        public Node Next
        {
            get { return next; }
            set { next = value; }
        }

        public T Data //T as return type of property
        {
            get { return data; }
            set { data = value; }
        }
    }

    public GenericList() //constructor
    {
        head = null;
    }

    public void AddHead(T t) //T as method parameter type
    {
        Node n = new Node(t);
        n.Next = head;
        head = n;
    }
}
```

```

// Implementation of the iterator
public System.Collections.Generic.IEnumerator<T> GetEnumerator()
{
    Node current = head;
    while (current != null)
    {
        yield return current.Data;
        current = current.Next;
    }
}

// IEnumerable<T> inherits from IEnumerable, therefore this class
// must implement both the generic and non-generic versions of
// GetEnumerator. In most cases, the non-generic method can
// simply call the generic method.
System.Collections.IEnumerator System.Collections.IEnumerable.GetEnumerator()
{
    return GetEnumerator();
}
}

public class SortedList<T> : GenericList<T> where T : System.IComparable<T>
{
    // A simple, unoptimized sort algorithm that
    // orders list elements from lowest to highest:

    public void BubbleSort()
    {
        if (null == head || null == head.Next)
        {
            return;
        }
        bool swapped;

        do
        {
            Node previous = null;
            Node current = head;
            swapped = false;

            while (current.next != null)
            {
                // Because we need to call this method, the SortedList
                // class is constrained on IComparable<T>
                if (current.Data.CompareTo(current.next.Data) > 0)
                {
                    Node tmp = current.next;
                    current.next = current.next.next;
                    tmp.next = current;

                    if (previous == null)
                    {
                        head = tmp;
                    }
                    else
                    {
                        previous.next = tmp;
                    }
                    previous = tmp;
                    swapped = true;
                }
                else
                {
                    previous = current;
                    current = current.next;
                }
            }
        } while (swapped);
    }
}

```

```

}

// A simple class that implements IComparable<T> using itself as the
// type argument. This is a common design pattern in objects that
// are stored in generic lists.
public class Person : System.IComparable<Person>
{
    string name;
    int age;

    public Person(string s, int i)
    {
        name = s;
        age = i;
    }

    // This will cause list elements to be sorted on age values.
    public int CompareTo(Person p)
    {
        return age - p.age;
    }

    public override string ToString()
    {
        return name + ":" + age;
    }

    // Must implement Equals.
    public bool Equals(Person p)
    {
        return (this.age == p.age);
    }
}

public class Program
{
    public static void Main()
    {
        //Declare and instantiate a new generic SortedList class.
        //Person is the type argument.
        SortedList<Person> list = new SortedList<Person>();

        //Create name and age values to initialize Person objects.
        string[] names = new string[]
        {
            "Franscoise",
            "Bill",
            "Li",
            "Sandra",
            "Gunnar",
            "Alok",
            "Hiroyuki",
            "Maria",
            "Alessandro",
            "Raul"
        };

        int[] ages = new int[] { 45, 19, 28, 23, 18, 9, 108, 72, 30, 35 };

        //Populate the list.
        for (int x = 0; x < 10; x++)
        {
            list.AddHead(new Person(names[x], ages[x]));
        }

        //Print out unsorted list.
        foreach (Person p in list)
        {
            System.Console.WriteLine(p.ToString());
        }
    }
}

```

```

    }
    System.Console.WriteLine("Done with unsorted list");

    //Sort the list.
    list.BubbleSort();

    //Print out sorted list.
    foreach (Person p in list)
    {
        System.Console.WriteLine(p.ToString());
    }
    System.Console.WriteLine("Done with sorted list");
}
}

```

Multiple interfaces can be specified as constraints on a single type, as follows:

```

class Stack<T> where T : System.IComparable<T>, IEnumerable<T>
{
}

```

An interface can define more than one type parameter, as follows:

```

interface IDictionary<K, V>
{
}

```

The rules of inheritance that apply to classes also apply to interfaces:

```

interface IMonth<T> { }

interface IJanuary    : IMonth<int> { } //No error
interface IFebruary<T> : IMonth<int> { } //No error
interface IMarch<T>    : IMonth<T> { }   //No error
//interface IApril<T>   : IMonth<T, U> { } //Error

```

Generic interfaces can inherit from non-generic interfaces if the generic interface is covariant, which means it only uses its type parameter as a return value. In the .NET class library, [IEnumerable<T>](#) inherits from [IEnumerable](#) because [IEnumerable<T>](#) only uses [T](#) in the return value of [GetEnumerator](#) and in the [Current](#) property getter.

Concrete classes can implement closed constructed interfaces, as follows:

```

interface IBaseInterface<T> { }

class SampleClass : IBaseInterface<string> { }

```

Generic classes can implement generic interfaces or closed constructed interfaces as long as the class parameter list supplies all arguments required by the interface, as follows:

```

interface IBaseInterface1<T> { }
interface IBaseInterface2<T, U> { }

class SampleClass1<T> : IBaseInterface1<T> { } //No error
class SampleClass2<T> : IBaseInterface2<T, string> { } //No error

```

The rules that control method overloading are the same for methods within generic classes, generic structs, or

generic interfaces. For more information, see [Generic Methods](#).

See also

- [C# Programming Guide](#)
- [Introduction to Generics](#)
- [interface](#)
- [Generics](#)

Generic Methods (C# Programming Guide)

12/28/2021 • 2 minutes to read • [Edit Online](#)

A generic method is a method that is declared with type parameters, as follows:

```
static void Swap<T>(ref T lhs, ref T rhs)
{
    T temp;
    temp = lhs;
    lhs = rhs;
    rhs = temp;
}
```

The following code example shows one way to call the method by using `int` for the type argument:

```
public static void TestSwap()
{
    int a = 1;
    int b = 2;

    Swap<int>(ref a, ref b);
    System.Console.WriteLine(a + " " + b);
}
```

You can also omit the type argument and the compiler will infer it. The following call to `Swap` is equivalent to the previous call:

```
Swap(ref a, ref b);
```

The same rules for type inference apply to static methods and instance methods. The compiler can infer the type parameters based on the method arguments you pass in; it cannot infer the type parameters only from a constraint or return value. Therefore type inference does not work with methods that have no parameters. Type inference occurs at compile time before the compiler tries to resolve overloaded method signatures. The compiler applies type inference logic to all generic methods that share the same name. In the overload resolution step, the compiler includes only those generic methods on which type inference succeeded.

Within a generic class, non-generic methods can access the class-level type parameters, as follows:

```
class SampleClass<T>
{
    void Swap(ref T lhs, ref T rhs) { }
}
```

If you define a generic method that takes the same type parameters as the containing class, the compiler generates warning [CS0693](#) because within the method scope, the argument supplied for the inner `T` hides the argument supplied for the outer `T`. If you require the flexibility of calling a generic class method with type arguments other than the ones provided when the class was instantiated, consider providing another identifier for the type parameter of the method, as shown in `GenericList2<T>` in the following example.

```

class GenericList<T>
{
    // CS0693
    void SampleMethod<T>() { }
}

class GenericList2<T>
{
    //No warning
    void SampleMethod<U>() { }
}

```

Use constraints to enable more specialized operations on type parameters in methods. This version of `Swap<T>`, now named `SwapIfGreater<T>`, can only be used with type arguments that implement [IComparable<T>](#).

```

void SwapIfGreater<T>(ref T lhs, ref T rhs) where T : System.IComparable<T>
{
    T temp;
    if (lhs.CompareTo(rhs) > 0)
    {
        temp = lhs;
        lhs = rhs;
        rhs = temp;
    }
}

```

Generic methods can be overloaded on several type parameters. For example, the following methods can all be located in the same class:

```

void DoWork() { }
void DoWork<T>() { }
void DoWork<T, U>() { }

```

C# Language Specification

For more information, see the [C# Language Specification](#).

See also

- [System.Collections.Generic](#)
- [C# Programming Guide](#)
- [Introduction to Generics](#)
- [Methods](#)

Generics and Arrays (C# Programming Guide)

12/28/2021 • 2 minutes to read • [Edit Online](#)

In C# 2.0 and later, single-dimensional arrays that have a lower bound of zero automatically implement [IList<T>](#). This enables you to create generic methods that can use the same code to iterate through arrays and other collection types. This technique is primarily useful for reading data in collections. The [IList<T>](#) interface cannot be used to add or remove elements from an array. An exception will be thrown if you try to call an [IList<T>](#) method such as [RemoveAt](#) on an array in this context.

The following code example demonstrates how a single generic method that takes an [IList<T>](#) input parameter can iterate through both a list and an array, in this case an array of integers.

```
class Program
{
    static void Main()
    {
        int[] arr = { 0, 1, 2, 3, 4 };
        List<int> list = new List<int>();

        for (int x = 5; x < 10; x++)
        {
            list.Add(x);
        }

        ProcessItems<int>(arr);
        ProcessItems<int>(list);
    }

    static void ProcessItems<T>(IList<T> coll)
    {
        // IsReadOnly returns True for the array and False for the List.
        System.Console.WriteLine
            ("IsReadOnly returns {0} for this collection.",
            coll.IsReadOnly);

        // The following statement causes a run-time exception for the
        // array, but not for the List.
        //coll.RemoveAt(4);

        foreach (T item in coll)
        {
            System.Console.Write(item.ToString() + " ");
        }
        System.Console.WriteLine();
    }
}
```

See also

- [System.Collections.Generic](#)
- [C# Programming Guide](#)
- [Generics](#)
- [Arrays](#)
- [Generics](#)

Generic Delegates (C# Programming Guide)

12/28/2021 • 2 minutes to read • [Edit Online](#)

A [delegate](#) can define its own type parameters. Code that references the generic delegate can specify the type argument to create a closed constructed type, just like when instantiating a generic class or calling a generic method, as shown in the following example:

```
public delegate void Del<T>(T item);
public static void Notify(int i) { }

Del<int> m1 = new Del<int>(Notify);
```

C# version 2.0 has a new feature called method group conversion, which applies to concrete as well as generic delegate types, and enables you to write the previous line with this simplified syntax:

```
Del<int> m2 = Notify;
```

Delegates defined within a generic class can use the generic class type parameters in the same way that class methods do.

```
class Stack<T>
{
    T[] items;
    int index;

    public delegate void StackDelegate(T[] items);
}
```

Code that references the delegate must specify the type argument of the containing class, as follows:

```
private static void DoWork(float[] items) { }

public static void TestStack()
{
    Stack<float> s = new Stack<float>();
    Stack<float>.StackDelegate d = DoWork;
}
```

Generic delegates are especially useful in defining events based on the typical design pattern because the sender argument can be strongly typed and no longer has to be cast to and from [Object](#).

```

delegate void StackEventHandler<T, U>(T sender, U eventArgs);

class Stack<T>
{
    public class StackEventArgs : System.EventArgs { }
    public event StackEventHandler<Stack<T>, StackEventArgs> stackEvent;

    protected virtual void OnStackChanged(StackEventArgs a)
    {
        stackEvent(this, a);
    }
}

class SampleClass
{
    public void HandleStackChange<T>(Stack<T> stack, Stack<T>.StackEventArgs args) { }
}

public static void Test()
{
    Stack<double> s = new Stack<double>();
    SampleClass o = new SampleClass();
    s.stackEvent += o.HandleStackChange;
}

```

See also

- [System.Collections.Generic](#)
- [C# Programming Guide](#)
- [Introduction to Generics](#)
- [Generic Methods](#)
- [Generic Classes](#)
- [Generic Interfaces](#)
- [Delegates](#)
- [Generics](#)

Differences Between C++ Templates and C# Generics (C# Programming Guide)

12/28/2021 • 2 minutes to read • [Edit Online](#)

C# Generics and C++ templates are both language features that provide support for parameterized types. However, there are many differences between the two. At the syntax level, C# generics are a simpler approach to parameterized types without the complexity of C++ templates. In addition, C# does not attempt to provide all of the functionality that C++ templates provide. At the implementation level, the primary difference is that C# generic type substitutions are performed at run time and generic type information is thereby preserved for instantiated objects. For more information, see [Generics in the Run Time](#).

The following are the key differences between C# Generics and C++ templates:

- C# generics do not provide the same amount of flexibility as C++ templates. For example, it is not possible to call arithmetic operators in a C# generic class, although it is possible to call user defined operators.
- C# does not allow non-type template parameters, such as `template C<int i> {}`.
- C# does not support explicit specialization; that is, a custom implementation of a template for a specific type.
- C# does not support partial specialization: a custom implementation for a subset of the type arguments.
- C# does not allow the type parameter to be used as the base class for the generic type.
- C# does not allow type parameters to have default types.
- In C#, a generic type parameter cannot itself be a generic, although constructed types can be used as generics. C++ does allow template parameters.
- C++ allows code that might not be valid for all type parameters in the template, which is then checked for the specific type used as the type parameter. C# requires code in a class to be written in such a way that it will work with any type that satisfies the constraints. For example, in C++ it is possible to write a function that uses the arithmetic operators `+` and `-` on objects of the type parameter, which will produce an error at the time of instantiation of the template with a type that does not support these operators. C# disallows this; the only language constructs allowed are those that can be deduced from the constraints.

See also

- [C# Programming Guide](#)
- [Introduction to Generics](#)
- [Templates](#)

Generics in the Run Time (C# Programming Guide)

12/28/2021 • 3 minutes to read • [Edit Online](#)

When a generic type or method is compiled into Microsoft intermediate language (MSIL), it contains metadata that identifies it as having type parameters. How the MSIL for a generic type is used differs based on whether the supplied type parameter is a value type or reference type.

When a generic type is first constructed with a value type as a parameter, the runtime creates a specialized generic type with the supplied parameter or parameters substituted in the appropriate locations in the MSIL. Specialized generic types are created one time for each unique value type that is used as a parameter.

For example, suppose your program code declared a stack that is constructed of integers:

```
Stack<int> stack;
```

At this point, the runtime generates a specialized version of the `Stack<T>` class that has the integer substituted appropriately for its parameter. Now, whenever your program code uses a stack of integers, the runtime reuses the generated specialized `Stack<T>` class. In the following example, two instances of a stack of integers are created, and they share a single instance of the `Stack<int>` code:

```
Stack<int> stackOne = new Stack<int>();  
Stack<int> stackTwo = new Stack<int>();
```

However, suppose that another `Stack<T>` class with a different value type such as a `long` or a user-defined structure as its parameter is created at another point in your code. As a result, the runtime generates another version of the generic type and substitutes a `long` in the appropriate locations in MSIL. Conversions are no longer necessary because each specialized generic class natively contains the value type.

Generics work somewhat differently for reference types. The first time a generic type is constructed with any reference type, the runtime creates a specialized generic type with object references substituted for the parameters in the MSIL. Then, every time that a constructed type is instantiated with a reference type as its parameter, regardless of what type it is, the runtime reuses the previously created specialized version of the generic type. This is possible because all references are the same size.

For example, suppose you had two reference types, a `Customer` class and an `Order` class, and also suppose that you created a stack of `Customer` types:

```
class Customer { }  
class Order { }
```

```
Stack<Customer> customers;
```

At this point, the runtime generates a specialized version of the `Stack<T>` class that stores object references that will be filled in later instead of storing data. Suppose the next line of code creates a stack of another reference type, which is named `Order`:

```
Stack<Order> orders = new Stack<Order>();
```


Unlike with value types, another specialized version of the [Stack<T>](#) class is not created for the `Order` type. Instead, an instance of the specialized version of the [Stack<T>](#) class is created and the `orders` variable is set to reference it. Suppose that you then encountered a line of code to create a stack of a `Customer` type:

```
customers = new Stack<Customer>();
```

As with the previous use of the [Stack<T>](#) class created by using the `Order` type, another instance of the specialized [Stack<T>](#) class is created. The pointers that are contained therein are set to reference an area of memory the size of a `Customer` type. Because the number of reference types can vary wildly from program to program, the C# implementation of generics greatly reduces the amount of code by reducing to one the number of specialized classes created by the compiler for generic classes of reference types.

Moreover, when a generic C# class is instantiated by using a value type or reference type parameter, reflection can query it at run time and both its actual type and its type parameter can be ascertained.

See also

- [System.Collections.Generic](#)
- [C# Programming Guide](#)
- [Introduction to Generics](#)
- [Generics](#)

Generics and Reflection (C# Programming Guide)

12/28/2021 • 2 minutes to read • [Edit Online](#)

Because the Common Language Runtime (CLR) has access to generic type information at run time, you can use reflection to obtain information about generic types in the same way as for non-generic types. For more information, see [Generics in the Run Time](#).

In .NET Framework 2.0, several new members were added to the [Type](#) class to enable run-time information for generic types. See the documentation on these classes for more information on how to use these methods and properties. The [System.Reflection.Emit](#) namespace also contains new members that support generics. See [How to: Define a Generic Type with Reflection Emit](#).

For a list of the invariant conditions for terms used in generic reflection, see the [IsGenericType](#) property remarks.

SYSTEM.TYPE MEMBER NAME	DESCRIPTION
IsGenericType	Returns true if a type is generic.
GetGenericArguments	Returns an array of Type objects that represent the type arguments supplied for a constructed type, or the type parameters of a generic type definition.
GetGenericTypeDefinition	Returns the underlying generic type definition for the current constructed type.
GetGenericParameterConstraints	Returns an array of Type objects that represent the constraints on the current generic type parameter.
ContainsGenericParameters	Returns true if the type or any of its enclosing types or methods contain type parameters for which specific types have not been supplied.
GenericParameterAttributes	Gets a combination of GenericParameterAttributes flags that describe the special constraints of the current generic type parameter.
GenericParameterPosition	For a Type object that represents a type parameter, gets the position of the type parameter in the type parameter list of the generic type definition or generic method definition that declared the type parameter.
IsGenericParameter	Gets a value that indicates whether the current Type represents a type parameter of a generic type or method definition.
IsGenericTypeDefinition	Gets a value that indicates whether the current Type represents a generic type definition, from which other generic types can be constructed. Returns true if the type represents the definition of a generic type.

SYSTEM.TYPE MEMBER NAME	DESCRIPTION
DeclaringMethod	Returns the generic method that defined the current generic type parameter, or null if the type parameter was not defined by a generic method.
MakeGenericType	Substitutes the elements of an array of types for the type parameters of the current generic type definition, and returns a Type object representing the resulting constructed type.

In addition, members of the [MethodInfo](#) class enable run-time information for generic methods. See the [IsGenericMethod](#) property remarks for a list of invariant conditions for terms used to reflect on generic methods.

SYSTEM.REFLECTION.MEMBERINFO MEMBER NAME	DESCRIPTION
IsGenericMethod	Returns true if a method is generic.
GetGenericArguments	Returns an array of Type objects that represent the type arguments of a constructed generic method or the type parameters of a generic method definition.
GetGenericMethodDefinition	Returns the underlying generic method definition for the current constructed method.
ContainsGenericParameters	Returns true if the method or any of its enclosing types contain any type parameters for which specific types have not been supplied.
IsGenericMethodDefinition	Returns true if the current MethodInfo represents the definition of a generic method.
MakeGenericMethod	Substitutes the elements of an array of types for the type parameters of the current generic method definition, and returns a MethodInfo object representing the resulting constructed method.

See also

- [C# Programming Guide](#)
- [Generics](#)
- [Reflection and Generic Types](#)
- [Generics](#)

Generics and Attributes (C# Programming Guide)

12/28/2021 • 2 minutes to read • [Edit Online](#)

Attributes can be applied to generic types in the same way as non-generic types. For more information on applying attributes, see [Attributes](#).

Custom attributes are only permitted to reference open generic types, which are generic types for which no type arguments are supplied, and closed constructed generic types, which supply arguments for all type parameters.

The following examples use this custom attribute:

```
class CustomAttribute : System.Attribute
{
    public System.Object info;
}
```

An attribute can reference an open generic type:

```
public class GenericClass1<T> { }

[CustomAttribute(info = typeof(GenericClass1<>))]
class ClassA { }
```

Specify multiple type parameters using the appropriate number of commas. In this example, `GenericClass2` has two type parameters:

```
public class GenericClass2<T, U> { }

[CustomAttribute(info = typeof(GenericClass2<,>))]
class ClassB { }
```

An attribute can reference a closed constructed generic type:

```
public class GenericClass3<T, U, V> { }

[CustomAttribute(info = typeof(GenericClass3<int, double, string>))]
class ClassC { }
```

An attribute that references a generic type parameter will cause a compile-time error:

```
//[CustomAttribute(info = typeof(GenericClass3<int, T, string>))] //Error
class ClassD<T> { }
```

A generic type cannot inherit from [Attribute](#):

```
//public class CustomAtt<T> : System.Attribute { } //Error
```

To obtain information about a generic type or type parameter at run time, you can use the methods of [System.Reflection](#). For more information, see [Generics and Reflection](#)

See also

- [C# Programming Guide](#)
- [Generics](#)
- [Attributes](#)

File system and the registry (C# Programming Guide)

12/28/2021 • 2 minutes to read • [Edit Online](#)

The following articles show how to use C# and .NET to perform various basic operations on files, folders, and the registry.

In this section

TITLE	DESCRIPTION
How to iterate through a directory tree	Shows how to manually iterate through a directory tree.
How to get information about files, folders, and drives	Shows how to retrieve information such as creation times and size, about files, folders and drives.
How to create a file or folder	Shows how to create a new file or folder.
How to copy, delete, and move files and folders (C# Programming Guide)	Shows how to copy, delete and move files and folders.
How to provide a progress dialog box for file operations	Shows how to display a standard Windows progress dialog for certain file operations.
How to write to a text file	Shows how to write to a text file.
How to read from a text file	Shows how to read from a text file.
How to read a text file one line at a time	Shows how to retrieve text from a file one line at a time.
How to create a key in the registry	Shows how to write a key to the system registry.

Related sections

- [File and Stream I/O](#)
- [How to copy, delete, and move files and folders \(C# Programming Guide\)](#)
- [C# Programming Guide](#)
- [System.IO](#)

How to iterate through a directory tree (C# Programming Guide)

12/28/2021 • 7 minutes to read • [Edit Online](#)

The phrase "iterate a directory tree" means to access each file in each nested subdirectory under a specified root folder, to any depth. You do not necessarily have to open each file. You can just retrieve the name of the file or subdirectory as a `string`, or you can retrieve additional information in the form of a [System.IO.FileInfo](#) or [System.IO.DirectoryInfo](#) object.

NOTE

In Windows, the terms "directory" and "folder" are used interchangeably. Most documentation and user interface text uses the term "folder," but .NET class libraries use the term "directory."

In the simplest case, in which you know for certain that you have access permissions for all directories under a specified root, you can use the `System.IO.SearchOption.AllDirectories` flag. This flag returns all the nested subdirectories that match the specified pattern. The following example shows how to use this flag.

```
root.GetDirectories("*.**", System.IO.SearchOption.AllDirectories);
```

The weakness in this approach is that if any one of the subdirectories under the specified root causes a [DirectoryNotFoundException](#) or [UnauthorizedAccessException](#), the whole method fails and returns no directories. The same is true when you use the [GetFiles](#) method. If you have to handle these exceptions on specific subfolders, you must manually walk the directory tree, as shown in the following examples.

When you manually walk a directory tree, you can handle the files first (*pre-order traversal*), or the subdirectories first (*post-order traversal*). If you perform a pre-order traversal, you visit files directly under that folder itself, and then walk the whole tree under the current folder. Post-order traversal is the other way around, walking the whole tree beneath before getting to the current folder's files. The examples later in this document perform pre-order traversal, but you can easily modify them to perform post-order traversal.

Another option is whether to use recursion or a stack-based traversal. The examples later in this document show both approaches.

If you have to perform a variety of operations on files and folders, you can modularize these examples by refactoring the operation into separate functions that you can invoke by using a single delegate.

NOTE

NTFS file systems can contain *reparse points* in the form of *junction points*, *symbolic links*, and *hard links*. .NET methods such as [GetFiles](#) and [GetDirectories](#) will not return any subdirectories under a reparse point. This behavior guards against the risk of entering into an infinite loop when two reparse points refer to each other. In general, you should use extreme caution when you deal with reparse points to ensure that you do not unintentionally modify or delete files. If you require precise control over reparse points, use platform invoke or native code to call the appropriate Win32 file system methods directly.

Examples

The following example shows how to walk a directory tree by using recursion. The recursive approach is elegant but has the potential to cause a stack overflow exception if the directory tree is large and deeply nested.

The particular exceptions that are handled, and the particular actions that are performed on each file or folder, are provided as examples only. You should modify this code to meet your specific requirements. See the comments in the code for more information.

```
public class RecursiveFileSearch
{
    static System.Collections.Specialized.StringCollection log = new
System.Collections.Specialized.StringCollection();

    static void Main()
    {
        // Start with drives if you have to search the entire computer.
        string[] drives = System.Environment.GetLogicalDrives();

        foreach (string dr in drives)
        {
            System.IO.DriveInfo di = new System.IO.DriveInfo(dr);

            // Here we skip the drive if it is not ready to be read. This
            // is not necessarily the appropriate action in all scenarios.
            if (!di.IsReady)
            {
                Console.WriteLine("The drive {0} could not be read", di.Name);
                continue;
            }
            System.IO.DirectoryInfo rootDir = di.RootDirectory;
            WalkDirectoryTree(rootDir);
        }

        // Write out all the files that could not be processed.
        Console.WriteLine("Files with restricted access:");
        foreach (string s in log)
        {
            Console.WriteLine(s);
        }
        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key");
        Console.ReadKey();
    }

    static void WalkDirectoryTree(System.IO.DirectoryInfo root)
    {
        System.IO.FileInfo[] files = null;
        System.IO.DirectoryInfo[] subDirs = null;

        // First, process all the files directly under this folder
        try
        {
            files = root.GetFiles("*.");
        }
        // This is thrown if even one of the files requires permissions greater
        // than the application provides.
        catch (UnauthorizedAccessException e)
        {
            // This code just writes out the message and continues to recurse.
            // You may decide to do something different here. For example, you
            // can try to elevate your privileges and access the file again.
            log.Add(e.Message);
        }

        catch (System.IO.DirectoryNotFoundException e)
        {
            Console.WriteLine(e.Message);
        }
    }
}
```



```

if (files != null)
{
    foreach (System.IO.FileInfo fi in files)
    {
        // In this example, we only access the existing FileInfo object. If we
        // want to open, delete or modify the file, then
        // a try-catch block is required here to handle the case
        // where the file has been deleted since the call to TraverseTree().
        Console.WriteLine(fi.FullName);
    }

    // Now find all the subdirectories under this directory.
    subDirs = root.GetDirectories();

    foreach (System.IO.DirectoryInfo dirInfo in subDirs)
    {
        // Recursive call for each subdirectory.
        WalkDirectoryTree(dirInfo);
    }
}
}
}

```

The following example shows how to iterate through files and folders in a directory tree without using recursion. This technique uses the generic [Stack<T>](#) collection type, which is a last in first out (LIFO) stack.

The particular exceptions that are handled, and the particular actions that are performed on each file or folder, are provided as examples only. You should modify this code to meet your specific requirements. See the comments in the code for more information.

```

public class StackBasedIteration
{
    static void Main(string[] args)
    {
        // Specify the starting folder on the command line, or in
        // Visual Studio in the Project > Properties > Debug pane.
        TraverseTree(args[0]);

        Console.WriteLine("Press any key");
        Console.ReadKey();
    }

    public static void TraverseTree(string root)
    {
        // Data structure to hold names of subfolders to be
        // examined for files.
        Stack<string> dirs = new Stack<string>(20);

        if (!System.IO.Directory.Exists(root))
        {
            throw new ArgumentException();
        }
        dirs.Push(root);

        while (dirs.Count > 0)
        {
            string currentDir = dirs.Pop();
            string[] subDirs;
            try
            {
                subDirs = System.IO.Directory.GetDirectories(currentDir);
            }
            // An UnauthorizedAccessException exception will be thrown if we do not have
            // discovery permission on a folder or file. It may or may not be acceptable
            // to ignore the exception and continue enumerating the remaining files and

```

```

// folders. It is also possible (but unlikely) that a DirectoryNotFoundException
// will be raised. This will happen if currentDir has been deleted by
// another application or thread after our call to Directory.Exists. The
// choice of which exceptions to catch depends entirely on the specific task
// you are intending to perform and also on how much you know with certainty
// about the systems on which this code will run.
catch (UnauthorizedAccessException e)
{
    Console.WriteLine(e.Message);
    continue;
}
catch (System.IO.DirectoryNotFoundException e)
{
    Console.WriteLine(e.Message);
    continue;
}

string[] files = null;
try
{
    files = System.IO.Directory.GetFiles(currentDir);
}

catch (UnauthorizedAccessException e)
{
    Console.WriteLine(e.Message);
    continue;
}

catch (System.IO.DirectoryNotFoundException e)
{
    Console.WriteLine(e.Message);
    continue;
}

// Perform the required action on each file here.
// Modify this block to perform your required task.
foreach (string file in files)
{
    try
    {
        // Perform whatever action is required in your scenario.
        System.IO.FileInfo fi = new System.IO.FileInfo(file);
        Console.WriteLine("{0}: {1}, {2}", fi.Name, fi.Length, fi.CreationTime);
    }
    catch (System.IO.FileNotFoundException e)
    {
        // If file was deleted by a separate application
        // or thread since the call to TraverseTree()
        // then just continue.
        Console.WriteLine(e.Message);
        continue;
    }
}

// Push the subdirectories onto the stack for traversal.
// This could also be done before handing the files.
foreach (string str in subDirs)
    dirs.Push(str);
}
}
}

```

It is generally too time-consuming to test every folder to determine whether your application has permission to open it. Therefore, the code example just encloses that part of the operation in a `try/catch` block. You can modify the `catch` block so that when you are denied access to a folder, you try to elevate your permissions and

then access it again. As a rule, only catch those exceptions that you can handle without leaving your application in an unknown state.

If you must store the contents of a directory tree, either in memory or on disk, the best option is to store only the [FullName](#) property (of type `string`) for each file. You can then use this string to create a new [FileInfo](#) or [DirectoryInfo](#) object as necessary, or open any file that requires additional processing.

Robust Programming

Robust file iteration code must take into account many complexities of the file system. For more information on the Windows file system, see [NTFS overview](#).

See also

- [System.IO](#)
- [LINQ and File Directories](#)
- [File System and the Registry \(C# Programming Guide\)](#)

How to get information about files, folders, and drives (C# Programming Guide)

12/28/2021 • 2 minutes to read • [Edit Online](#)

In .NET, you can access file system information by using the following classes:

- [System.IO.FileInfo](#)
- [System.IO.DirectoryInfo](#)
- [System.IO.DriveInfo](#)
- [System.IO.Directory](#)
- [System.IO.File](#)

The [FileInfo](#) and [DirectoryInfo](#) classes represent a file or directory and contain properties that expose many of the file attributes that are supported by the NTFS file system. They also contain methods for opening, closing, moving, and deleting files and folders. You can create instances of these classes by passing a string that represents the name of the file, folder, or drive in to the constructor:

```
System.IO.DriveInfo di = new System.IO.DriveInfo(@"C:\");
```

You can also obtain the names of files, folders, or drives by using calls to [DirectoryInfo.GetDirectories](#), [DirectoryInfo.GetFiles](#), and [DriveInfo.RootDirectory](#).

The [System.IO.Directory](#) and [System.IO.File](#) classes provide static methods for retrieving information about directories and files.

Example

The following example shows various ways to access information about files and folders.

```
class FileSysInfo
{
    static void Main()
    {
        // You can also use System.Environment.GetLogicalDrives to
        // obtain names of all logical drives on the computer.
        System.IO.DriveInfo di = new System.IO.DriveInfo(@"C:\");
        Console.WriteLine(di.TotalFreeSpace);
        Console.WriteLine(di.VolumeLabel);

        // Get the root directory and print out some information about it.
        System.IO.DirectoryInfo dirInfo = di.RootDirectory;
        Console.WriteLine(dirInfo.Attributes.ToString());

        // Get the files in the directory and print out some information about them.
        System.IO.FileInfo[] fileNames = dirInfo.GetFiles("*.");

        foreach (System.IO.FileInfo fi in fileNames)
        {
            Console.WriteLine("{0}: {1}: {2}", fi.Name, fi.LastAccessTime, fi.Length);
        }

        // Get the subdirectories directly that is under the root.
```

```

// See "How to: Iterate Through a Directory Tree" for an example of how to
// iterate through an entire tree.
System.IO.DirectoryInfo[] dirInfos = dirInfo.GetDirectories("*.");

foreach (System.IO.DirectoryInfo d in dirInfos)
{
    Console.WriteLine(d.Name);
}

// The Directory and File classes provide several static methods
// for accessing files and directories.

// Get the current application directory.
string currentDirName = System.IO.Directory.GetCurrentDirectory();
Console.WriteLine(currentDirName);

// Get an array of file names as strings rather than FileInfo objects.
// Use this method when storage space is an issue, and when you might
// hold on to the file name reference for a while before you try to access
// the file.
string[] files = System.IO.Directory.GetFiles(currentDirName, "*.txt");

foreach (string s in files)
{
    // Create the FileInfo object only when needed to ensure
    // the information is as current as possible.
    System.IO.FileInfo fi = null;
    try
    {
        fi = new System.IO.FileInfo(s);
    }
    catch (System.IO.FileNotFoundException e)
    {
        // To inform the user and continue is
        // sufficient for this demonstration.
        // Your application may require different behavior.
        Console.WriteLine(e.Message);
        continue;
    }
    Console.WriteLine("{0} : {1}", fi.Name, fi.Directory);
}

// Change the directory. In this case, first check to see
// whether it already exists, and create it if it does not.
// If this is not appropriate for your application, you can
// handle the System.IO.IOException that will be raised if the
// directory cannot be found.
if (!System.IO.Directory.Exists(@"C:\Users\Public\TestFolder\"))
{
    System.IO.Directory.CreateDirectory(@"C:\Users\Public\TestFolder\");
}

System.IO.Directory.SetCurrentDirectory(@"C:\Users\Public\TestFolder\");

currentDirName = System.IO.Directory.GetCurrentDirectory();
Console.WriteLine(currentDirName);

// Keep the console window open in debug mode.
Console.WriteLine("Press any key to exit.");
Console.ReadKey();
}
}

```

Robust Programming

When you process user-specified path strings, you should also handle exceptions for the following conditions:

- The file name is malformed. For example, it contains invalid characters or only white space.
- The file name is null.
- The file name is longer than the system-defined maximum length.
- The file name contains a colon (:).

If the application does not have sufficient permissions to read the specified file, the `Exists` method returns `false` regardless of whether a path exists; the method does not throw an exception.

See also

- [System.IO](#)
- [C# Programming Guide](#)
- [File System and the Registry \(C# Programming Guide\)](#)

How to create a file or folder (C# Programming Guide)

12/28/2021 • 3 minutes to read • [Edit Online](#)

You can programmatically create a folder on your computer, create a subfolder, create a file in the subfolder, and write data to the file.

Example

```
public class CreateFileOrFolder
{
    static void Main()
    {
        // Specify a name for your top-level folder.
        string folderName = @"c:\Top-Level Folder";

        // To create a string that specifies the path to a subfolder under your
        // top-level folder, add a name for the subfolder to folderName.
        string pathString = System.IO.Path.Combine(folderName, "SubFolder");

        // You can write out the path name directly instead of using the Combine
        // method. Combine just makes the process easier.
        string pathString2 = @"c:\Top-Level Folder\SubFolder2";

        // You can extend the depth of your path if you want to.
        //pathString = System.IO.Path.Combine(pathString, "SubSubFolder");

        // Create the subfolder. You can verify in File Explorer that you have this
        // structure in the C: drive.
        //      Local Disk (C:)
        //          Top-Level Folder
        //              SubFolder
        System.IO.Directory.CreateDirectory(pathString);

        // Create a file name for the file you want to create.
        string fileName = System.IO.Path.GetRandomFileName();

        // This example uses a random string for the name, but you also can specify
        // a particular name.
        //string fileName = "MyNewFile.txt";

        // Use Combine again to add the file name to the path.
        pathString = System.IO.Path.Combine(pathString, fileName);

        // Verify the path that you have constructed.
        Console.WriteLine("Path to my file: {0}\n", pathString);

        // Check that the file doesn't already exist. If it doesn't exist, create
        // the file and write integers 0 - 99 to it.
        // DANGER: System.IO.File.Create will overwrite the file if it already exists.
        // This could happen even with random file names, although it is unlikely.
        if (!System.IO.File.Exists(pathString))
        {
            using (System.IO.FileStream fs = System.IO.File.Create(pathString))
            {
                for (byte i = 0; i < 100; i++)
                {
                    fs.WriteByte(i);
                }
            }
        }
    }
}
```

```

    }
    else
    {
        Console.WriteLine("File \"{0}\" already exists.", fileName);
        return;
    }

    // Read and display the data from your file.
    try
    {
        byte[] readBuffer = System.IO.File.ReadAllBytes(pathString);
        foreach (byte b in readBuffer)
        {
            Console.Write(b + " ");
        }
        Console.WriteLine();
    }
    catch (System.IO.IOException e)
    {
        Console.WriteLine(e.Message);
    }

    // Keep the console window open in debug mode.
    System.Console.WriteLine("Press any key to exit.");
    System.Console.ReadKey();
}
// Sample output:

// Path to my file: c:\Top-Level Folder\SubFolder\ttxvauxe.vv0

//0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29
//30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56
// 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80 81 82 8
//3 84 85 86 87 88 89 90 91 92 93 94 95 96 97 98 99
}

```

If the folder already exists, [CreateDirectory](#) does nothing, and no exception is thrown. However, [File.Create](#) replaces an existing file with a new file. The example uses an `if - else` statement to prevent an existing file from being replaced.

By making the following changes in the example, you can specify different outcomes based on whether a file with a certain name already exists. If such a file doesn't exist, the code creates one. If such a file exists, the code appends data to that file.

- Specify a non-random file name.

```

// Comment out the following line.
//string fileName = System.IO.Path.GetRandomFileName();

// Replace that line with the following assignment.
string fileName = "MyNewFile.txt";

```

- Replace the `if - else` statement with the `using` statement in the following code.

```

using (System.IO.FileStream fs = new System.IO.FileStream(pathString, FileMode.Append))
{
    for (byte i = 0; i < 100; i++)
    {
        fs.WriteByte(i);
    }
}

```


Run the example several times to verify that data is added to the file each time.

For more `FileMode` values that you can try, see [FileMode](#).

The following conditions may cause an exception:

- The folder name is malformed. For example, it contains illegal characters or is only white space ([ArgumentException](#) class). Use the [Path](#) class to create valid path names.
- The parent folder of the folder to be created is read-only ([IOException](#) class).
- The folder name is `null` ([ArgumentNullException](#) class).
- The folder name is too long ([PathTooLongException](#) class).
- The folder name is only a colon, ":" ([PathTooLongException](#) class).

.NET Security

An instance of the [SecurityException](#) class may be thrown in partial-trust situations.

If you don't have permission to create the folder, the example throws an instance of the [UnauthorizedAccessException](#) class.

See also

- [System.IO](#)
- [C# Programming Guide](#)
- [File System and the Registry \(C# Programming Guide\)](#)

How to copy, delete, and move files and folders (C# Programming Guide)

12/28/2021 • 3 minutes to read • [Edit Online](#)

The following examples show how to copy, move, and delete files and folders in a synchronous manner by using the [System.IO.File](#), [System.IO.Directory](#), [System.IO.FileInfo](#), and [System.IO.DirectoryInfo](#) classes from the [System.IO](#) namespace. These examples do not provide a progress bar or any other user interface. If you want to provide a standard progress dialog box, see [How to provide a progress dialog box for file operations](#).

Use [System.IO.FileSystemWatcher](#) to provide events that will enable you to calculate the progress when operating on multiple files. Another approach is to use platform invoke to call the relevant file-related methods in the Windows Shell. For information about how to perform these file operations asynchronously, see [Asynchronous File I/O](#).

Examples

The following example shows how to copy files and directories.

```

// Simple synchronous file copy operations with no user interface.
// To run this sample, first create the following directories and files:
// C:\Users\Public\TestFolder
// C:\Users\Public\TestFolder\test.txt
// C:\Users\Public\TestFolder\SubDir\test.txt
public class SimpleFileCopy
{
    static void Main()
    {
        string fileName = "test.txt";
        string sourcePath = @"C:\Users\Public\TestFolder";
        string targetPath = @"C:\Users\Public\TestFolder\SubDir";

        // Use Path class to manipulate file and directory paths.
        string sourceFile = System.IO.Path.Combine(sourcePath, fileName);
        string destFile = System.IO.Path.Combine(targetPath, fileName);

        // To copy a folder's contents to a new location:
        // Create a new target folder.
        // If the directory already exists, this method does not create a new directory.
        System.IO.Directory.CreateDirectory(targetPath);

        // To copy a file to another location and
        // overwrite the destination file if it already exists.
        System.IO.File.Copy(sourceFile, destFile, true);

        // To copy all the files in one directory to another directory.
        // Get the files in the source folder. (To recursively iterate through
        // all subfolders under the current directory, see
        // "How to: Iterate Through a Directory Tree.")
        // Note: Check for target path was performed previously
        //      in this code example.
        if (System.IO.Directory.Exists(sourcePath))
        {
            string[] files = System.IO.Directory.GetFiles(sourcePath);

            // Copy the files and overwrite destination files if they already exist.
            foreach (string s in files)
            {
                // Use static Path methods to extract only the file name from the path.
                fileName = System.IO.Path.GetFileName(s);
                destFile = System.IO.Path.Combine(targetPath, fileName);
                System.IO.File.Copy(s, destFile, true);
            }
        }
        else
        {
            Console.WriteLine("Source path does not exist!");
        }

        // Keep console window open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}

```

The following example shows how to move files and directories.

```
// Simple synchronous file move operations with no user interface.
public class SimpleFileMove
{
    static void Main()
    {
        string sourceFile = @"C:\Users\Public\public\test.txt";
        string destinationFile = @"C:\Users\Public\private\test.txt";

        // To move a file or folder to a new location:
        System.IO.File.Move(sourceFile, destinationFile);

        // To move an entire directory. To programmatically modify or combine
        // path strings, use the System.IO.Path class.
        System.IO.Directory.Move(@"C:\Users\Public\public\test\", @"C:\Users\Public\private");
    }
}
```

The following example shows how to delete files and directories.

```
// Simple synchronous file deletion operations with no user interface.
// To run this sample, create the following files on your drive:
// C:\Users\Public\DeleteTest\test1.txt
// C:\Users\Public\DeleteTest\test2.txt
// C:\Users\Public\DeleteTest\SubDir\test2.txt

public class SimpleFileDelete
{
    static void Main()
    {
        // Delete a file by using File class static method...
        if(System.IO.File.Exists(@"C:\Users\Public\DeleteTest\test.txt"))
        {
            // Use a try block to catch IOExceptions, to
            // handle the case of the file already being
            // opened by another process.
            try
            {
                System.IO.File.Delete(@"C:\Users\Public\DeleteTest\test.txt");
            }
            catch (System.IO.IOException e)
            {
                Console.WriteLine(e.Message);
                return;
            }
        }

        // ...or by using FileInfo instance method.
        System.IO.FileInfo fi = new System.IO.FileInfo(@"C:\Users\Public\DeleteTest\test2.txt");
        try
        {
            fi.Delete();
        }
        catch (System.IO.IOException e)
        {
            Console.WriteLine(e.Message);
        }

        // Delete a directory. Must be writable or empty.
        try
        {
            System.IO.Directory.Delete(@"C:\Users\Public\DeleteTest");
        }
        catch (System.IO.IOException e)
        {
            Console.WriteLine(e.Message);
        }
    }
}
```

```

// Delete a directory and all subdirectories with Directory static method...
if(System.IO.Directory.Exists(@"C:\Users\Public\DeleteTest"))
{
    try
    {
        System.IO.Directory.Delete(@"C:\Users\Public\DeleteTest", true);
    }

    catch (System.IO.IOException e)
    {
        Console.WriteLine(e.Message);
    }
}

// ...or with DirectoryInfo instance method.
System.IO.DirectoryInfo di = new System.IO.DirectoryInfo(@"C:\Users\Public\public");
// Delete this dir and all subdirs.
try
{
    {
        di.Delete(true);
    }
}
catch (System.IO.IOException e)
{
    Console.WriteLine(e.Message);
}
}
}

```

See also

- [System.IO](#)
- [C# Programming Guide](#)
- [File System and the Registry \(C# Programming Guide\)](#)
- [How to provide a progress dialog box for file operations](#)
- [File and Stream I/O](#)
- [Common I/O Tasks](#)

How to provide a progress dialog box for file operations (C# Programming Guide)

12/28/2021 • 2 minutes to read • [Edit Online](#)

You can provide a standard dialog box that shows progress on file operations in Windows if you use the [CopyFile\(String, String, UIOption\)](#) method in the [Microsoft.VisualBasic](#) namespace.

NOTE

Your computer might show different names or locations for some of the Visual Studio user interface elements in the following instructions. The Visual Studio edition that you have and the settings that you use determine these elements. For more information, see [Personalizing the IDE](#).

To add a reference in Visual Studio

1. On the menu bar, choose **Project, Add Reference**.

The **Reference Manager** dialog box appears.

2. In the **Assemblies** area, choose **Framework** if it isn't already chosen.
3. In the list of names, select the **Microsoft.VisualBasic** check box, and then choose the **OK** button to close the dialog box.

Example

The following code copies the directory that `sourcePath` specifies into the directory that `destinationPath` specifies. This code also provides a standard dialog box that shows the estimated amount of time remaining before the operation finishes.

```
// The following using directive requires a project reference to Microsoft.VisualBasic.
using Microsoft.VisualBasic.FileIO;

class FileProgress
{
    static void Main()
    {
        // Specify the path to a folder that you want to copy. If the folder is small,
        // you won't have time to see the progress dialog box.
        string sourcePath = @"C:\Windows\symbols\";
        // Choose a destination for the copied files.
        string destinationPath = @"C:\TestFolder";

        FileSystem.CopyDirectory(sourcePath, destinationPath,
                                UIOption.AllDialogs);
    }
}
```

See also

- [File System and the Registry \(C# Programming Guide\)](#)

How to write to a text file (C# Programming Guide)

12/28/2021 • 2 minutes to read • [Edit Online](#)

In this article, there are several examples showing various ways to write text to a file. The first two examples use static convenience methods on the [System.IO.File](#) class to write each element of any `IEnumerable<string>` and a `string` to a text file. The third example shows how to add text to a file when you have to process each line individually as you write to the file. In the first three examples, you overwrite all existing content in the file. The final example shows how to append text to an existing file.

These examples all write string literals to files. If you want to format text written to a file, use the [Format](#) method or C# [string interpolation](#) feature.

Write a collection of strings to a file

```
using System.IO;
using System.Threading.Tasks;

class WriteAllLines
{
    public static async Task ExampleAsync()
    {
        string[] lines =
        {
            "First line", "Second line", "Third line"
        };

        await File.WriteAllLinesAsync("WriteLines.txt", lines);
    }
}
```

The preceding source code example:

- Instantiates a string array with three values.
- Awaits a call to [File.WriteAllLinesAsync](#) which:
 - Asynchronously creates a file name *WriteLines.txt*. If the file already exists, it is overwritten.
 - Writes the given lines to the file.
 - Closes the file, automatically flushing and disposing as needed.

Write one string to a file

```

using System.IO;
using System.Threading.Tasks;

class WriteAllText
{
    public static async Task ExampleAsync()
    {
        string text =
            "A class is the most powerful data type in C#. Like a structure, " +
            "a class defines the data and behavior of the data type. ";

        await File.WriteAllTextAsync("WriteText.txt", text);
    }
}

```

The preceding source code example:

- Instantiates a string given the assigned string literal.
- Awaits a call to [File.WriteAllTextAsync](#) which:
 - Asynchronously creates a file name *WriteText.txt*. If the file already exists, it is overwritten.
 - Writes the given text to the file.
 - Closes the file, automatically flushing and disposing as needed.

Write selected strings from an array to a file

```

using System.IO;
using System.Threading.Tasks;

class StreamWriterOne
{
    public static async Task ExampleAsync()
    {
        string[] lines = { "First line", "Second line", "Third line" };
        using StreamWriter file = new("WriteLines2.txt");

        foreach (string line in lines)
        {
            if (!line.Contains("Second"))
            {
                await file.WriteLineAsync(line);
            }
        }
    }
}

```

The preceding source code example:

- Instantiates a string array with three values.
- Instantiates a [StreamWriter](#) with a file path of *WriteLines2.txt* as a [using declaration](#).
- Iterates through all the lines.
- Conditionally awaits a call to [StreamWriter.WriteLineAsync\(String\)](#), which writes the line to the file when the line doesn't contain `"Second"`.

Append text to an existing file


```
using System.IO;
using System.Threading.Tasks;

class StreamWriterTwo
{
    public static async Task ExampleAsync()
    {
        using StreamWriter file = new("WriteLines2.txt", append: true);
        await file.WriteLineAsync("Fourth line");
    }
}
```

The preceding source code example:

- Instantiates a string array with three values.
- Instantiates a [StreamWriter](#) with a file path of *WriteLines2.txt* as a [using declaration](#), passing in `true` to append.
- Awaits a call to [StreamWriter.WriteLineAsync\(String\)](#), which writes the string to the file as an appended line.

Exceptions

The following conditions may cause an exception:

- [InvalidOperationException](#): The file exists and is read-only.
- [PathTooLongException](#): The path name may be too long.
- [IOException](#): The disk may be full.

There are additional conditions that may cause exceptions when working with the file system, it is best to program defensively.

See also

- [C# Programming Guide](#)
- [File System and the Registry \(C# Programming Guide\)](#)

How to read from a text file (C# Programming Guide)

12/28/2021 • 2 minutes to read • [Edit Online](#)

This example reads the contents of a text file by using the static methods [ReadAllText](#) and [ReadAllLines](#) from the [System.IO.File](#) class.

For an example that uses [StreamReader](#), see [How to read a text file one line at a time](#).

NOTE

The files that are used in this example are created in the topic [How to write to a text file](#).

Example

```
class ReadFromFile
{
    static void Main()
    {
        // The files used in this example are created in the topic
        // How to: Write to a Text File. You can change the path and
        // file name to substitute text files of your own.

        // Example #1
        // Read the file as one string.
        string text = System.IO.File.ReadAllText(@"C:\Users\Public\TestFolder\WriteText.txt");

        // Display the file contents to the console. Variable text is a string.
        System.Console.WriteLine("Contents of WriteText.txt = {0}", text);

        // Example #2
        // Read each line of the file into a string array. Each element
        // of the array is one line of the file.
        string[] lines = System.IO.File.ReadAllLines(@"C:\Users\Public\TestFolder\WriteLines2.txt");

        // Display the file contents by using a foreach loop.
        System.Console.WriteLine("Contents of Writelines2.txt = ");
        foreach (string line in lines)
        {
            // Use a tab to indent each line of the file.
            Console.WriteLine("\t" + line);
        }

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit.");
        System.Console.ReadKey();
    }
}
```

Compiling the Code

Copy the code and paste it into a C# console application.

If you are not using the text files from [How to write to a text file](#), replace the argument to `ReadAllText` and `ReadAllLines` with the appropriate path and file name on your computer.

Robust Programming

The following conditions may cause an exception:

- The file doesn't exist or doesn't exist at the specified location. Check the path and the spelling of the file name.

.NET Security

Do not rely on the name of a file to determine the contents of the file. For example, the file `myFile.cs` might not be a C# source file.

See also

- [System.IO](#)
- [C# Programming Guide](#)
- [File System and the Registry \(C# Programming Guide\)](#)

How to read a text file one line at a time (C# Programming Guide)

12/28/2021 • 2 minutes to read • [Edit Online](#)

This example reads the contents of a text file, one line at a time, into a string using the `ReadLines` method of the `File` class. Each text line is stored into the string `line` and displayed on the screen.

Example

```
int counter = 0;

// Read the file and display it line by line.
foreach (string line in System.IO.File.ReadLines(@"c:\test.txt"))
{
    System.Console.WriteLine(line);
    counter++;
}

System.Console.WriteLine("There were {0} lines.", counter);
// Suspend the screen.
System.Console.ReadLine();
```

Compiling the Code

Copy the code and paste it into the `Main` method of a console application.

Replace `"c:\test.txt"` with the actual file name.

Robust Programming

The following conditions may cause an exception:

- The file may not exist.

.NET Security

Do not make decisions about the contents of the file based on the name of the file. For example, the file `myFile.cs` may not be a C# source file.

See also

- [System.IO](#)
- [C# Programming Guide](#)
- [File System and the Registry \(C# Programming Guide\)](#)

How to create a key in the registry (C# Programming Guide)

12/28/2021 • 2 minutes to read • [Edit Online](#)

This example adds the value pair, "Name" and "Isabella", to the current user's registry, under the key "Names".

Example

```
Microsoft.Win32.RegistryKey key;  
key = Microsoft.Win32.Registry.CurrentUser.CreateSubKey("Names");  
key.SetValue("Name", "Isabella");  
key.Close();
```

Compiling the Code

- Copy the code and paste it into the `Main` method of a console application.
- Replace the `Names` parameter with the name of a key that exists directly under the `HKEY_CURRENT_USER` node of the registry.
- Replace the `Name` parameter with the name of a value that exists directly under the `Names` node.

Robust Programming

Examine the registry structure to find a suitable location for your key. For example, you might want to open the `Software` key of the current user, and create a key with your company's name. Then add the registry values to your company's key.

The following conditions might cause an exception:

- The name of the key is null.
- The user does not have permissions to create registry keys.
- The key name exceeds the 255-character limit.
- The key is closed.
- The registry key is read-only.

.NET Security

It is more secure to write data to the user folder — `Microsoft.Win32.Registry.CurrentUser` — rather than to the local computer — `Microsoft.Win32.Registry.LocalMachine`.

When you create a registry value, you need to decide what to do if that value already exists. Another process, perhaps a malicious one, may have already created the value and have access to it. When you put data in the registry value, the data is available to the other process. To prevent this, use the.

`Overload:Microsoft.Win32.RegistryKey.GetValue` method. It returns null if the key does not already exist.

It is not secure to store secrets, such as passwords, in the registry as plain text, even if the registry key is protected by access control lists (ACL).

See also

- [System.IO](#)
- [C# Programming Guide](#)
- [File System and the Registry \(C# Programming Guide\)](#)
- [Read, write and delete from the registry with C#](#)

Interoperability (C# Programming Guide)

12/28/2021 • 2 minutes to read • [Edit Online](#)

Interoperability enables you to preserve and take advantage of existing investments in unmanaged code. Code that runs under the control of the common language runtime (CLR) is called *managed code*, and code that runs outside the CLR is called *unmanaged code*. COM, COM+, C++ components, ActiveX components, and Microsoft Windows API are examples of unmanaged code.

.NET enables interoperability with unmanaged code through platform invoke services, the [System.Runtime.InteropServices](#) namespace, C++ interoperability, and COM interoperability (COM interop).

In This Section

[Interoperability Overview](#)

Describes methods to interoperate between C# managed code and unmanaged code.

[How to access Office interop objects by using C# features](#)

Describes features that are introduced in Visual C# to facilitate Office programming.

[How to use indexed properties in COM interop programming](#)

Describes how to use indexed properties to access COM properties that have parameters.

[How to use platform invoke to play a WAV file](#)

Describes how to use platform invoke services to play a .wav sound file on the Windows operating system.

[Walkthrough: Office Programming](#)

Shows how to create an Excel workbook and a Word document that contains a link to the workbook.

[Example COM Class](#)

Demonstrates how to expose a C# class as a COM object.

C# Language Specification

For more information, see [Basic concepts](#) in the [C# Language Specification](#). The language specification is the definitive source for C# syntax and usage.

See also

- [Marshal.ReleaseComObject](#)
- [C# Programming Guide](#)
- [Interoperating with Unmanaged Code](#)
- [Walkthrough: Office Programming](#)

Interoperability Overview (C# Programming Guide)

12/28/2021 • 3 minutes to read • [Edit Online](#)

The topic describes methods to enable interoperability between C# managed code and unmanaged code.

Platform Invoke

Platform invoke is a service that enables managed code to call unmanaged functions that are implemented in dynamic link libraries (DLLs), such as those in the Microsoft Windows API. It locates and invokes an exported function and marshals its arguments (integers, strings, arrays, structures, and so on) across the interoperation boundary as needed.

For more information, see [Consuming Unmanaged DLL Functions](#) and [How to use platform invoke to play a WAV file](#).

NOTE

The [Common Language Runtime](#) (CLR) manages access to system resources. Calling unmanaged code that is outside the CLR bypasses this security mechanism, and therefore presents a security risk. For example, unmanaged code might call resources in unmanaged code directly, bypassing CLR security mechanisms. For more information, see [Security in .NET](#).

C++ Interop

You can use C++ interop, also known as It Just Works (IJW), to wrap a native C++ class so that it can be consumed by code that is authored in C# or another .NET language. To do this, you write C++ code to wrap a native DLL or COM component. Unlike other .NET languages, Visual C++ has interoperability support that enables managed and unmanaged code to be located in the same application and even in the same file. You then build the C++ code by using the `/clr` compiler switch to produce a managed assembly. Finally, you add a reference to the assembly in your C# project and use the wrapped objects just as you would use other managed classes.

Exposing COM Components to C#

You can consume a COM component from a C# project. The general steps are as follows:

1. Locate a COM component to use and register it. Use `regsvr32.exe` to register or un-register a COM DLL.
2. Add to the project a reference to the COM component or type library.

When you add the reference, Visual Studio uses the [Tlbimp.exe \(Type Library Importer\)](#), which takes a type library as input, to output a .NET interop assembly. The assembly, also named a runtime callable wrapper (RCW), contains managed classes and interfaces that wrap the COM classes and interfaces that are in the type library. Visual Studio adds to the project a reference to the generated assembly.

3. Create an instance of a class that is defined in the RCW. This, in turn, creates an instance of the COM object.
4. Use the object just as you use other managed objects. When the object is reclaimed by garbage collection, the instance of the COM object is also released from memory.

For more information, see [Exposing COM Components to the .NET Framework](#).

Exposing C# to COM

COM clients can consume C# types that have been correctly exposed. The basic steps to expose C# types are as follows:

1. Add interop attributes in the C# project.

You can make an assembly COM visible by modifying Visual C# project properties. For more information, see [Assembly Information Dialog Box](#).

2. Generate a COM type library and register it for COM usage.

You can modify Visual C# project properties to automatically register the C# assembly for COM interop. Visual Studio uses the [Regasm.exe \(Assembly Registration Tool\)](#), using the `/t1b` command-line switch, which takes a managed assembly as input, to generate a type library. This type library describes the `public` types in the assembly and adds registry entries so that COM clients can create managed classes.

For more information, see [Exposing .NET Framework Components to COM](#) and [Example COM Class](#).

See also

- [Improving Interop Performance](#)
- [Introduction to Interoperability between COM and .NET](#)
- [Introduction to COM Interop in Visual Basic](#)
- [Marshaling between Managed and Unmanaged Code](#)
- [Interoperating with Unmanaged Code](#)
- [C# Programming Guide](#)

How to access Office interop objects (C# Programming Guide)

12/28/2021 • 12 minutes to read • [Edit Online](#)

C# has features that simplify access to Office API objects. The new features include named and optional arguments, a new type called `dynamic`, and the ability to pass arguments to reference parameters in COM methods as if they were value parameters.

In this topic you will use the new features to write code that creates and displays a Microsoft Office Excel worksheet. You will then write code to add an Office Word document that contains an icon that is linked to the Excel worksheet.

To complete this walkthrough, you must have Microsoft Office Excel 2007 and Microsoft Office Word 2007, or later versions, installed on your computer.

NOTE

Your computer might show different names or locations for some of the Visual Studio user interface elements in the following instructions. The Visual Studio edition that you have and the settings that you use determine these elements. For more information, see [Personalizing the IDE](#).

To create a new console application

1. Start Visual Studio.
2. On the **File** menu, point to **New**, and then click **Project**. The **New Project** dialog box appears.
3. In the **Installed Templates** pane, expand **Visual C#**, and then click **Windows**.
4. Look at the top of the **New Project** dialog box to make sure that **.NET Framework 4** (or later version) is selected as a target framework.
5. In the **Templates** pane, click **Console Application**.
6. Type a name for your project in the **Name** field.
7. Click **OK**.

The new project appears in **Solution Explorer**.

To add references

1. In **Solution Explorer**, right-click your project's name and then click **Add Reference**. The **Add Reference** dialog box appears.
2. On the **Assemblies** page, select **Microsoft.Office.Interop.Word** in the **Component Name** list, and then hold down the CTRL key and select **Microsoft.Office.Interop.Excel**. If you do not see the assemblies, you may need to ensure they are installed and displayed. See [How to: Install Office Primary Interop Assemblies](#).
3. Click **OK**.

To add necessary using directives

1. In **Solution Explorer**, right-click the *Program.cs* file and then click **View Code**.
2. Add the following `using` directives to the top of the code file:

```
using Excel = Microsoft.Office.Interop.Excel;
using Word = Microsoft.Office.Interop.Word;
```

To create a list of bank accounts

1. Paste the following class definition into **Program.cs**, under the `Program` class.

```
public class Account
{
    public int ID { get; set; }
    public double Balance { get; set; }
}
```

2. Add the following code to the `Main` method to create a `bankAccounts` list that contains two accounts.

```
// Create a list of accounts.
var bankAccounts = new List<Account> {
    new Account {
        ID = 345678,
        Balance = 541.27
    },
    new Account {
        ID = 1230221,
        Balance = -127.44
    }
};
```

To declare a method that exports account information to Excel

1. Add the following method to the `Program` class to set up an Excel worksheet.

Method `Add` has an optional parameter for specifying a particular template. Optional parameters, new in C# 4, enable you to omit the argument for that parameter if you want to use the parameter's default value. Because no argument is sent in the following code, `Add` uses the default template and creates a new workbook. The equivalent statement in earlier versions of C# requires a placeholder argument:

```
ExcelApp.Workbooks.Add(Type.Missing) .
```

```
static void DisplayInExcel(IEnumerable<Account> accounts)
{
    var excelApp = new Excel.Application();
    // Make the object visible.
    excelApp.Visible = true;

    // Create a new, empty workbook and add it to the collection returned
    // by property Workbooks. The new workbook becomes the active workbook.
    // Add has an optional parameter for specifying a particular template.
    // Because no argument is sent in this example, Add creates a new workbook.
    excelApp.Workbooks.Add();

    // This example uses a single workSheet. The explicit type casting is
    // removed in a later procedure.
    Excel._Worksheet workSheet = (Excel.Worksheet)excelApp.ActiveSheet;
}
```

2. Add the following code at the end of `DisplayInExcel`. The code inserts values into the first two columns of the first row of the worksheet.

```
// Establish column headings in cells A1 and B1.
workSheet.Cells[1, "A"] = "ID Number";
workSheet.Cells[1, "B"] = "Current Balance";
```

3. Add the following code at the end of `DisplayInExcel`. The `foreach` loop puts the information from the list of accounts into the first two columns of successive rows of the worksheet.

```
var row = 1;
foreach (var acct in accounts)
{
    row++;
    workSheet.Cells[row, "A"] = acct.ID;
    workSheet.Cells[row, "B"] = acct.Balance;
}
```

4. Add the following code at the end of `DisplayInExcel` to adjust the column widths to fit the content.

```
workSheet.Columns[1].AutoFit();
workSheet.Columns[2].AutoFit();
```

Earlier versions of C# require explicit casting for these operations because `ExcelApp.Columns[1]` returns an `Object`, and `AutoFit` is an Excel [Range](#) method. The following lines show the casting.

```
((Excel.Range)workSheet.Columns[1]).AutoFit();
((Excel.Range)workSheet.Columns[2]).AutoFit();
```

C# 4, and later versions, converts the returned `Object` to `dynamic` automatically if the assembly is referenced by the [EmbedInteropTypes](#) compiler option or, equivalently, if the Excel **Embed Interop Types** property is set to true. True is the default value for this property.

To run the project

1. Add the following line at the end of `Main`.

```
// Display the list in an Excel spreadsheet.  
DisplayInExcel(bankAccounts);
```

2. Press CTRL+F5.

An Excel worksheet appears that contains the data from the two accounts.

To add a Word document

1. To illustrate additional ways in which C# 4, and later versions, enhances Office programming, the following code opens a Word application and creates an icon that links to the Excel worksheet.

Paste method `CreateIconInWordDoc`, provided later in this step, into the `Program` class.

`CreateIconInWordDoc` uses named and optional arguments to reduce the complexity of the method calls to `Add` and `PasteSpecial`. These calls incorporate two other new features introduced in C# 4 that simplify calls to COM methods that have reference parameters. First, you can send arguments to the reference parameters as if they were value parameters. That is, you can send values directly, without creating a variable for each reference parameter. The compiler generates temporary variables to hold the argument values, and discards the variables when you return from the call. Second, you can omit the `ref` keyword in the argument list.

The `Add` method has four reference parameters, all of which are optional. In C# 4.0 and later versions, you can omit arguments for any or all of the parameters if you want to use their default values. In C# 3.0 and earlier versions, an argument must be provided for each parameter, and the argument must be a variable because the parameters are reference parameters.

The `PasteSpecial` method inserts the contents of the Clipboard. The method has seven reference parameters, all of which are optional. The following code specifies arguments for two of them: `Link`, to create a link to the source of the Clipboard contents, and `DisplayAsIcon`, to display the link as an icon. In C# 4.0 and later versions, you can use named arguments for those two and omit the others. Although these are reference parameters, you do not have to use the `ref` keyword, or to create variables to send in as arguments. You can send the values directly. In C# 3.0 and earlier versions, you must supply a variable argument for each reference parameter.

```
static void CreateIconInWordDoc()  
{  
    var wordApp = new Word.Application();  
    wordApp.Visible = true;  
  
    // The Add method has four reference parameters, all of which are  
    // optional. Visual C# allows you to omit arguments for them if  
    // the default values are what you want.  
    wordApp.Documents.Add();  
  
    // PasteSpecial has seven reference parameters, all of which are  
    // optional. This example uses named arguments to specify values  
    // for two of the parameters. Although these are reference  
    // parameters, you do not need to use the ref keyword, or to create  
    // variables to send in as arguments. You can send the values directly.  
    wordApp.Selection.PasteSpecial( Link: true, DisplayAsIcon: true);  
}
```

In C# 3.0 and earlier versions of the language, the following more complex code is required.

```

static void CreateIconInWordDoc2008()
{
    var wordApp = new Word.Application();
    wordApp.Visible = true;

    // The Add method has four parameters, all of which are optional.
    // In Visual C# 2008 and earlier versions, an argument has to be sent
    // for every parameter. Because the parameters are reference
    // parameters of type object, you have to create an object variable
    // for the arguments that represents 'no value'.

    object useDefaultValue = Type.Missing;

    wordApp.Documents.Add(ref useDefaultValue, ref useDefaultValue,
        ref useDefaultValue, ref useDefaultValue);

    // PasteSpecial has seven reference parameters, all of which are
    // optional. In this example, only two of the parameters require
    // specified values, but in Visual C# 2008 an argument must be sent
    // for each parameter. Because the parameters are reference parameters,
    // you have to construct variables for the arguments.
    object link = true;
    object displayAsIcon = true;

    wordApp.Selection.PasteSpecial( ref useDefaultValue,
                                    ref link,
                                    ref useDefaultValue,
                                    ref displayAsIcon,
                                    ref useDefaultValue,
                                    ref useDefaultValue,
                                    ref useDefaultValue);
}

```

2. Add the following statement at the end of `Main`.

```

// Create a Word document that contains an icon that links to
// the spreadsheet.
CreateIconInWordDoc();

```

3. Add the following statement at the end of `DisplayInExcel`. The `Copy` method adds the worksheet to the Clipboard.

```

// Put the spreadsheet contents on the clipboard. The Copy method has one
// optional parameter for specifying a destination. Because no argument
// is sent, the destination is the Clipboard.
workSheet.Range["A1:B3"].Copy();

```

4. Press CTRL+F5.

A Word document appears that contains an icon. Double-click the icon to bring the worksheet to the foreground.

To set the Embed Interop Types property

1. Additional enhancements are possible when you call a COM type that does not require a primary interop assembly (PIA) at run time. Removing the dependency on PIAs results in version independence and easier deployment. For more information about the advantages of programming without PIAs, see [Walkthrough: Embedding Types from Managed Assemblies](#).

In addition, programming is easier because the types that are required and returned by COM methods

can be represented by using the type `dynamic` instead of `object`. Variables that have type `dynamic` are not evaluated until run time, which eliminates the need for explicit casting. For more information, see [Using Type dynamic](#).

In C# 4, embedding type information instead of using PIAs is default behavior. Because of that default, several of the previous examples are simplified because explicit casting is not required. For example, the declaration of `worksheet` in `DisplayInExcel` is written as

`Excel._Worksheet workSheet = excelApp.ActiveSheet` rather than `Excel._Worksheet workSheet = (Excel.Worksheet)excelApp.ActiveSheet`. The calls to `AutoFit` in the same method also would require explicit casting without the default, because `ExcelApp.Columns[1]` returns an `object`, and `AutoFit` is an Excel method. The following code shows the casting.

```
((Excel.Range)workSheet.Columns[1]).AutoFit();  
((Excel.Range)workSheet.Columns[2]).AutoFit();
```

2. To change the default and use PIAs instead of embedding type information, expand the **References** node in **Solution Explorer** and then select **Microsoft.Office.Interop.Excel** or **Microsoft.Office.Interop.Word**.
3. If you cannot see the **Properties** window, press **F4**.
4. Find **Embed Interop Types** in the list of properties, and change its value to **False**. Equivalently, you can compile by using the **References** compiler option instead of **EmbedInteropTypes** at a command prompt.

To add additional formatting to the table

1. Replace the two calls to `AutoFit` in `DisplayInExcel` with the following statement.

```
// Call to AutoFormat in Visual C# 2010.  
workSheet.Range["A1", "B3"].AutoFormat(  
    Excel.XlRangeAutoFormat.xlRangeAutoFormatClassic2);
```

The **AutoFormat** method has seven value parameters, all of which are optional. Named and optional arguments enable you to provide arguments for none, some, or all of them. In the previous statement, an argument is supplied for only one of the parameters, `Format`. Because `Format` is the first parameter in the parameter list, you do not have to provide the parameter name. However, the statement might be easier to understand if the parameter name is included, as is shown in the following code.

```
// Call to AutoFormat in Visual C# 2010.  
workSheet.Range["A1", "B3"].AutoFormat(Format:  
    Excel.XlRangeAutoFormat.xlRangeAutoFormatClassic2);
```

2. Press **CTRL+F5** to see the result. Other formats are listed in the **XlRangeAutoFormat** enumeration.
3. Compare the statement in step 1 with the following code, which shows the arguments that are required in C# 3.0 and earlier versions.

```
// The AutoFormat method has seven optional value parameters. The
// following call specifies a value for the first parameter, and uses
// the default values for the other six.

// Call to AutoFormat in Visual C# 2008. This code is not part of the
// current solution.
excelApp.get_Range("A1", "B4").AutoFormat(Excel.XlRangeAutoFormat.xlRangeAutoFormatTable3,
    Type.Missing, Type.Missing, Type.Missing, Type.Missing, Type.Missing,
    Type.Missing);
```

Example

The following code shows the complete example.

```
using System;
using System.Collections.Generic;
using System.Linq;
using Excel = Microsoft.Office.Interop.Excel;
using Word = Microsoft.Office.Interop.Word;

namespace OfficeProgramminWalkthruComplete
{
    class Walkthrough
    {
        static void Main(string[] args)
        {
            // Create a list of accounts.
            var bankAccounts = new List<Account>
            {
                new Account {
                    ID = 345678,
                    Balance = 541.27
                },
                new Account {
                    ID = 1230221,
                    Balance = -127.44
                }
            };

            // Display the list in an Excel spreadsheet.
            DisplayInExcel(bankAccounts);

            // Create a Word document that contains an icon that links to
            // the spreadsheet.
            CreateIconInWordDoc();
        }

        static void DisplayInExcel(IEnumerable<Account> accounts)
        {
            var excelApp = new Excel.Application();
            // Make the object visible.
            excelApp.Visible = true;

            // Create a new, empty workbook and add it to the collection returned
            // by property Workbooks. The new workbook becomes the active workbook.
            // Add has an optional parameter for specifying a particular template.
            // Because no argument is sent in this example, Add creates a new workbook.
            excelApp.Workbooks.Add();

            // This example uses a single worksheet.
            Excel._Worksheet workSheet = excelApp.ActiveSheet;

            // Earlier versions of C# require explicit casting.
            //Excel._Worksheet workSheet = (Excel.Worksheet)excelApp.ActiveSheet;
```



```

// Establish column headings in cells A1 and B1.
workSheet.Cells[1, "A"] = "ID Number";
workSheet.Cells[1, "B"] = "Current Balance";

var row = 1;
foreach (var acct in accounts)
{
    row++;
    workSheet.Cells[row, "A"] = acct.ID;
    workSheet.Cells[row, "B"] = acct.Balance;
}

workSheet.Columns[1].AutoFit();
workSheet.Columns[2].AutoFit();

// Call to AutoFormat in Visual C#. This statement replaces the
// two calls to AutoFit.
workSheet.Range["A1", "B3"].AutoFormat(
    Excel.XlRangeAutoFormat.xlRangeAutoFormatClassic2);

// Put the spreadsheet contents on the clipboard. The Copy method has one
// optional parameter for specifying a destination. Because no argument
// is sent, the destination is the Clipboard.
workSheet.Range["A1:B3"].Copy();
}

static void CreateIconInWordDoc()
{
    var wordApp = new Word.Application();
    wordApp.Visible = true;

    // The Add method has four reference parameters, all of which are
    // optional. Visual C# allows you to omit arguments for them if
    // the default values are what you want.
    wordApp.Documents.Add();

    // PasteSpecial has seven reference parameters, all of which are
    // optional. This example uses named arguments to specify values
    // for two of the parameters. Although these are reference
    // parameters, you do not need to use the ref keyword, or to create
    // variables to send in as arguments. You can send the values directly.
    wordApp.Selection.PasteSpecial(Link: true, DisplayAsIcon: true);
}

}

public class Account
{
    public int ID { get; set; }
    public double Balance { get; set; }
}
}

```

See also

- [Type.Missing](#)
- [dynamic](#)
- [Using Type dynamic](#)
- [Named and Optional Arguments](#)
- [How to use named and optional arguments in Office programming](#)

How to use indexed properties in COM interop programming (C# Programming Guide)

12/28/2021 • 2 minutes to read • [Edit Online](#)

Indexed properties improve the way in which COM properties that have parameters are consumed in C# programming. Indexed properties work together with other features in Visual C#, such as [named and optional arguments](#), a new type ([dynamic](#)), and [embedded type information](#), to enhance Microsoft Office programming.

In earlier versions of C#, methods are accessible as properties only if the `get` method has no parameters and the `set` method has one and only one value parameter. However, not all COM properties meet those restrictions. For example, the Excel `Range[]` property has a `get` accessor that requires a parameter for the name of the range. In the past, because you could not access the `Range` property directly, you had to use the `get_Range` method instead, as shown in the following example.

```
// Visual C# 2008 and earlier.
var excelApp = new Excel.Application();
// . . .
Excel.Range targetRange = excelApp.get_Range("A1", Type.Missing);
```

Indexed properties enable you to write the following instead:

```
// Visual C# 2010.
var excelApp = new Excel.Application();
// . . .
Excel.Range targetRange = excelApp.Range["A1"];
```

NOTE

The previous example also uses the [optional arguments](#) feature, which enables you to omit `Type.Missing`.

Similarly to set the value of the `Value` property of a `Range` object in C# 3.0 and earlier, two arguments are required. One supplies an argument for an optional parameter that specifies the type of the range value. The other supplies the value for the `Value` property. The following examples illustrate these techniques. Both set the value of the A1 cell to `Name`.

```
// Visual C# 2008.
targetRange.SetValue(Type.Missing, "Name");
// Or
targetRange.Value2 = "Name";
```

Indexed properties enable you to write the following code instead.

```
// Visual C# 2010.
targetRange.Value = "Name";
```

You cannot create indexed properties of your own. The feature only supports consumption of existing indexed properties.

Example

The following code shows a complete example. For more information about how to set up a project that accesses the Office API, see [How to access Office interop objects by using C# features](#).

```
// You must add a reference to Microsoft.Office.Interop.Excel to run
// this example.
using System;
using Excel = Microsoft.Office.Interop.Excel;

namespace IndexedProperties
{
    class Program
    {
        static void Main(string[] args)
        {
            CSharp2010();
            //CSharp2008();
        }

        static void CSharp2010()
        {
            var excelApp = new Excel.Application();
            excelApp.Workbooks.Add();
            excelApp.Visible = true;

            Excel.Range targetRange = excelApp.Range["A1"];
            targetRange.Value = "Name";
        }

        static void CSharp2008()
        {
            var excelApp = new Excel.Application();
            excelApp.Workbooks.Add(Type.Missing);
            excelApp.Visible = true;

            Excel.Range targetRange = excelApp.get_Range("A1", Type.Missing);
            targetRange.set_Value(Type.Missing, "Name");
            // Or
            //targetRange.Value2 = "Name";
        }
    }
}
```

See also

- [Named and Optional Arguments](#)
- [dynamic](#)
- [Using Type dynamic](#)
- [How to use named and optional arguments in Office programming](#)
- [How to access Office interop objects by using C# features](#)
- [Walkthrough: Office Programming](#)

How to use platform invoke to play a WAV file (C# Programming Guide)

12/28/2021 • 2 minutes to read • [Edit Online](#)

The following C# code example illustrates how to use platform invoke services to play a WAV sound file on the Windows operating system.

Example

This example code uses [DllImportAttribute](#) to import `winmm.dll`'s `PlaySound` method entry point as `Form1.PlaySound()`. The example has a simple Windows Form with a button. Clicking the button opens a standard windows [OpenFileDialog](#) dialog box so that you can open a file to play. When a wave file is selected, it is played by using the `PlaySound()` method of the `winmm.dll` library. For more information about this method, see [Using the PlaySound function with Waveform-Audio Files](#). Browse and select a file that has a .wav extension, and then click **Open** to play the wave file by using platform invoke. A text box shows the full path of the file selected.

The **Open Files** dialog box is filtered to show only files that have a .wav extension through the filter settings:

```
dialog1.Filter = "Wav Files (*.wav)|*.wav";
```

```

using System.Windows.Forms;
using System.Runtime.InteropServices;

namespace WinSound
{
    public partial class Form1 : Form
    {
        private TextBox textBox1;
        private Button button1;

        public Form1() // Constructor.
        {
            InitializeComponent();
        }

        [DllImport("winmm.DLL", EntryPoint = "PlaySound", SetLastError = true, CharSet = CharSet.Unicode,
        ThrowOnUnmappableChar = true)]
        private static extern bool PlaySound(string szSound, System.IntPtr hMod, PlaySoundFlags flags);

        [System.Flags]
        public enum PlaySoundFlags : int
        {
            SND_SYNC = 0x0000,
            SND_ASYNC = 0x0001,
            SND_NODEFAULT = 0x0002,
            SND_LOOP = 0x0008,
            SND_NOSTOP = 0x0010,
            SND_NOWAIT = 0x00002000,
            SND_FILENAME = 0x00020000,
            SND_RESOURCE = 0x00040004
        }

        private void button1_Click(object sender, System.EventArgs e)
        {
            var dialog1 = new OpenFileDialog();

            dialog1.Title = "Browse to find sound file to play";
            dialog1.InitialDirectory = @"c:\";
            dialog1.Filter = "Wav Files (*.wav)|*.wav";
            dialog1.FilterIndex = 2;
            dialog1.RestoreDirectory = true;

            if (dialog1.ShowDialog() == DialogResult.OK)
            {
                textBox1.Text = dialog1.FileName;
                PlaySound(dialog1.FileName, new System.IntPtr(), PlaySoundFlags.SND_SYNC);
            }
        }

        private void Form1_Load(object sender, EventArgs e)
        {
            // Including this empty method in the sample because in the IDE,
            // when users click on the form, generates code that looks for a default method
            // with this name. We add it here to prevent confusion for those using the samples.
        }
    }
}

```

Compiling the code

1. Create a new C# Windows Forms Application project in Visual Studio and name it **WinSound**.
2. Copy the code above, and paste it over the contents of the *Form1.cs* file.
3. Copy the following code, and paste it in the *Form1.Designer.cs* file, in the `InitializeComponent()` method,

after any existing code.

```
this.button1 = new System.Windows.Forms.Button();
this.textBox1 = new System.Windows.Forms.TextBox();
this.SuspendLayout();
//
// button1
//
this.button1.Location = new System.Drawing.Point(192, 40);
this.button1.Name = "button1";
this.button1.Size = new System.Drawing.Size(88, 24);
this.button1.TabIndex = 0;
this.button1.Text = "Browse";
this.button1.Click += new System.EventHandler(this.button1_Click);
//
// textBox1
//
this.textBox1.Location = new System.Drawing.Point(8, 40);
this.textBox1.Name = "textBox1";
this.textBox1.Size = new System.Drawing.Size(168, 20);
this.textBox1.TabIndex = 1;
this.textBox1.Text = "File path";
//
// Form1
//
this.AutoScaleDimensions = new System.Drawing.SizeF(5, 13);
this.ClientSize = new System.Drawing.Size(292, 266);
this.Controls.Add(this.textBox1);
this.Controls.Add(this.button1);
this.Name = "Form1";
this.Text = "Platform Invoke WinSound C#";
this.ResumeLayout(false);
this.PerformLayout();
```

4. Compile and run the code.

See also

- [C# Programming Guide](#)
- [Interoperability Overview](#)
- [A Closer Look at Platform Invoke](#)
- [Marshaling Data with Platform Invoke](#)

Walkthrough: Office Programming (C# and Visual Basic)

12/28/2021 • 11 minutes to read • [Edit Online](#)

Visual Studio offers features in C# and Visual Basic that improve Microsoft Office programming. Helpful C# features include named and optional arguments and return values of type `dynamic`. In COM programming, you can omit the `ref` keyword and gain access to indexed properties. Features in Visual Basic include auto-implemented properties, statements in lambda expressions, and collection initializers.

Both languages enable embedding of type information, which allows deployment of assemblies that interact with COM components without deploying primary interop assemblies (PIAs) to the user's computer. For more information, see [Walkthrough: Embedding Types from Managed Assemblies](#).

This walkthrough demonstrates these features in the context of Office programming, but many of these features are also useful in general programming. In the walkthrough, you use an Excel Add-in application to create an Excel workbook. Next, you create a Word document that contains a link to the workbook. Finally, you see how to enable and disable the PIA dependency.

Prerequisites

You must have Microsoft Office Excel and Microsoft Office Word installed on your computer to complete this walkthrough.

NOTE

Your computer might show different names or locations for some of the Visual Studio user interface elements in the following instructions. The Visual Studio edition that you have and the settings that you use determine these elements. For more information, see [Personalizing the IDE](#).

To set up an Excel Add-in application

1. Start Visual Studio.
2. On the **File** menu, point to **New**, and then click **Project**.
3. In the **Installed Templates** pane, expand **Visual Basic** or **Visual C#**, expand **Office**, and then click the version year of the Office product.
4. In the **Templates** pane, click **Excel <version> Add-in**.
5. Look at the top of the **Templates** pane to make sure that **.NET Framework 4**, or a later version, appears in the **Target Framework** box.
6. Type a name for your project in the **Name** box, if you want to.
7. Click **OK**.
8. The new project appears in **Solution Explorer**.

To add references

1. In **Solution Explorer**, right-click your project's name and then click **Add Reference**. The **Add Reference** dialog box appears.

2. On the **Assemblies** tab, select **Microsoft.Office.Interop.Excel**, version `<version>.0.0.0` (for a key to the Office product version numbers, see [Microsoft Versions](#)), in the **Component Name** list, and then hold down the CTRL key and select **Microsoft.Office.Interop.Word**, `version <version>.0.0.0`. If you do not see the assemblies, you may need to ensure they are installed and displayed (see [How to: Install Office Primary Interop Assemblies](#)).
3. Click **OK**.

To add necessary Imports statements or using directives

1. In **Solution Explorer**, right-click the **ThisAddIn.vb** or **ThisAddIn.cs** file and then click **View Code**.
2. Add the following `Imports` statements (Visual Basic) or `using` directives (C#) to the top of the code file if they are not already present.

```
using System.Collections.Generic;
using Excel = Microsoft.Office.Interop.Excel;
using Word = Microsoft.Office.Interop.Word;
```

```
Imports Microsoft.Office.Interop
```

To create a list of bank accounts

1. In **Solution Explorer**, right-click your project's name, click **Add**, and then click **Class**. Name the class **Account.vb** if you are using Visual Basic or **Account.cs** if you are using C#. Click **Add**.
2. Replace the definition of the `Account` class with the following code. The class definitions use *auto-implemented properties*. For more information, see [Auto-Implemented Properties](#).

```
class Account
{
    public int ID { get; set; }
    public double Balance { get; set; }
}
```

```
Public Class Account
    Property ID As Integer = -1
    Property Balance As Double
End Class
```

3. To create a `bankAccounts` list that contains two accounts, add the following code to the `ThisAddIn_Startup` method in *ThisAddIn.vb* or *ThisAddIn.cs*. The list declarations use *collection initializers*. For more information, see [Collection Initializers](#).

```
var bankAccounts = new List<Account>
{
    new Account
    {
        ID = 345,
        Balance = 541.27
    },
    new Account
    {
        ID = 123,
        Balance = -127.44
    }
};
```



```
Dim bankAccounts As New List(Of Account) From {
    New Account With {
        .ID = 345,
        .Balance = 541.27
    },
    New Account With {
        .ID = 123,
        .Balance = -127.44
    }
}
```

To export data to Excel

1. In the same file, add the following method to the `ThisAddIn` class. The method sets up an Excel workbook and exports data to it.

```
void DisplayInExcel(IEnumerable<Account> accounts,
    Action<Account, Excel.Range> DisplayFunc)
{
    var excelApp = this.Application;
    // Add a new Excel workbook.
    excelApp.Workbooks.Add();
    excelApp.Visible = true;
    excelApp.Range["A1"].Value = "ID";
    excelApp.Range["B1"].Value = "Balance";
    excelApp.Range["A2"].Select();

    foreach (var ac in accounts)
    {
        DisplayFunc(ac, excelApp.ActiveCell);
        excelApp.ActiveCell.Offset[1, 0].Select();
    }
    // Copy the results to the Clipboard.
    excelApp.Range["A1:B3"].Copy();
}
```

```
Sub DisplayInExcel(ByVal accounts As IEnumerable(Of Account),
    ByVal DisplayAction As Action(Of Account, Excel.Range))

    With Me.Application
        ' Add a new Excel workbook.
        .Workbooks.Add()
        .Visible = True
        .Range("A1").Value = "ID"
        .Range("B1").Value = "Balance"
        .Range("A2").Select()

        For Each ac In accounts
            DisplayAction(ac, .ActiveCell)
            .ActiveCell.Offset(1, 0).Select()
        Next

        ' Copy the results to the Clipboard.
        .Range("A1:B3").Copy()
    End With
End Sub
```

Two new C# features are used in this method. Both of these features already exist in Visual Basic.

- Method `Add` has an *optional parameter* for specifying a particular template. Optional parameters, new in C# 4, enable you to omit the argument for that parameter if you want to use the parameter's default value. Because no argument is sent in the previous example, `Add` uses the

default template and creates a new workbook. The equivalent statement in earlier versions of C# requires a placeholder argument: `excelApp.Workbooks.Add(Type.Missing)`.

For more information, see [Named and Optional Arguments](#).

- The `Range` and `Offset` properties of the `Range` object use the *indexed properties* feature. This feature enables you to consume these properties from COM types by using the following typical C# syntax. Indexed properties also enable you to use the `Value` property of the `Range` object, eliminating the need to use the `Value2` property. The `Value` property is indexed, but the index is optional. Optional arguments and indexed properties work together in the following example.

```
// Visual C# 2010 provides indexed properties for COM programming.
excelApp.Range["A1"].Value = "ID";
excelApp.ActiveCell.Offset[1, 0].Select();
```

In earlier versions of the language, the following special syntax is required.

```
// In Visual C# 2008, you cannot access the Range, Offset, and Value
// properties directly.
excelApp.get_Range("A1").Value2 = "ID";
excelApp.ActiveCell.get_Offset(1, 0).Select();
```

You cannot create indexed properties of your own. The feature only supports consumption of existing indexed properties.

For more information, see [How to use indexed properties in COM interop programming](#).

2. Add the following code at the end of `DisplayInExcel` to adjust the column widths to fit the content.

```
excelApp.Columns[1].AutoFit();
excelApp.Columns[2].AutoFit();
```

```
' Add the following two lines at the end of the With statement.
.Columns(1).AutoFit()
.Columns(2).AutoFit()
```

These additions demonstrate another feature in C#: treating `Object` values returned from COM hosts such as Office as if they have type `dynamic`. This happens automatically when **Embed Interop Types** is set to its default value, `True`, or, equivalently, when the assembly is referenced by the **EmbedInteropTypes** compiler option. Type `dynamic` allows late binding, already available in Visual Basic, and avoids the explicit casting required in C# 3.0 and earlier versions of the language.

For example, `excelApp.Columns[1]` returns an `Object`, and `AutoFit` is an Excel `Range` method. Without `dynamic`, you must cast the object returned by `excelApp.Columns[1]` as an instance of `Range` before calling method `AutoFit`.

```
// Casting is required in Visual C# 2008.
((Excel.Range)excelApp.Columns[1]).AutoFit();

// Casting is not required in Visual C# 2010.
excelApp.Columns[1].AutoFit();
```

For more information about embedding interop types, see procedures "To find the PIA reference" and "To restore the PIA dependency" later in this topic. For more information about `dynamic`, see [dynamic](#) or

To invoke DisplayInExcel

1. Add the following code at the end of the `ThisAddIn_StartUp` method. The call to `DisplayInExcel` contains two arguments. The first argument is the name of the list of accounts to be processed. The second argument is a multiline lambda expression that defines how the data is to be processed. The `ID` and `balance` values for each account are displayed in adjacent cells, and the row is displayed in red if the balance is less than zero. For more information, see [Lambda Expressions](#).

```
DisplayInExcel(bankAccounts, (account, cell) =>
// This multiline lambda expression sets custom processing rules
// for the bankAccounts.
{
    cell.Value = account.ID;
    cell.Offset[0, 1].Value = account.Balance;
    if (account.Balance < 0)
    {
        cell.Interior.Color = 255;
        cell.Offset[0, 1].Interior.Color = 255;
    }
});
```

```
DisplayInExcel(bankAccounts,
    Sub(account, cell)
        ' This multiline lambda expression sets custom
        ' processing rules for the bankAccounts.
        cell.Value = account.ID
        cell.Offset(0, 1).Value = account.Balance

        If account.Balance < 0 Then
            cell.Interior.Color = RGB(255, 0, 0)
            cell.Offset(0, 1).Interior.Color = RGB(255, 0, 0)
        End If
    End Sub)
```

2. To run the program, press F5. An Excel worksheet appears that contains the data from the accounts.

To add a Word document

1. Add the following code at the end of the `ThisAddIn_StartUp` method to create a Word document that contains a link to the Excel workbook.

```
var wordApp = new Word.Application();
wordApp.Visible = true;
wordApp.Documents.Add();
wordApp.Selection.PasteSpecial(Link: true, DisplayAsIcon: true);
```

```
Dim wordApp As New Word.Application
wordApp.Visible = True
wordApp.Documents.Add()
wordApp.Selection.PasteSpecial(Link:=True, DisplayAsIcon:=True)
```

This code demonstrates several of the new features in C#: the ability to omit the `ref` keyword in COM programming, named arguments, and optional arguments. These features already exist in Visual Basic. The [PasteSpecial](#) method has seven parameters, all of which are defined as optional reference parameters. Named and optional arguments enable you to designate the parameters you want to access by name and to send arguments to only those parameters. In this example, arguments are sent to

indicate that a link to the workbook on the Clipboard should be created (parameter `Link`) and that the link is to be displayed in the Word document as an icon (parameter `DisplayAsIcon`). Visual C# also enables you to omit the `ref` keyword for these arguments.

To run the application

1. Press F5 to run the application. Excel starts and displays a table that contains the information from the two accounts in `bankAccounts`. Then a Word document appears that contains a link to the Excel table.

To clean up the completed project

1. In Visual Studio, click **Clean Solution** on the **Build** menu. Otherwise, the add-in will run every time that you open Excel on your computer.

To find the PIA reference

1. Run the application again, but do not click **Clean Solution**.
2. Select the **Start**. Locate **Microsoft Visual Studio <version>** and open a developer command prompt.
3. Type `ildasm` in the Developer Command Prompt for Visual Studio window, and then press ENTER. The IL DASM window appears.
4. On the **File** menu in the IL DASM window, select **File > Open**. Double-click **Visual Studio <version>**, and then double-click **Projects**. Open the folder for your project, and look in the bin/Debug folder for *your project name.dll*. Double-click *your project name.dll*. A new window displays your project's attributes, in addition to references to other modules and assemblies. Note that namespaces `Microsoft.Office.Interop.Excel` and `Microsoft.Office.Interop.Word` are included in the assembly. By default in Visual Studio, the compiler imports the types you need from a referenced PIA into your assembly.

For more information, see [How to: View Assembly Contents](#).

5. Double-click the **MANIFEST** icon. A window appears that contains a list of assemblies that contain items referenced by the project. `Microsoft.Office.Interop.Excel` and `Microsoft.Office.Interop.Word` are not included in the list. Because the types your project needs have been imported into your assembly, references to a PIA are not required. This makes deployment easier. The PIAs do not have to be present on the user's computer, and because an application does not require deployment of a specific version of a PIA, applications can be designed to work with multiple versions of Office, provided that the necessary APIs exist in all versions.

Because deployment of PIAs is no longer necessary, you can create an application in advanced scenarios that works with multiple versions of Office, including earlier versions. However, this works only if your code does not use any APIs that are not available in the version of Office you are working with. It is not always clear whether a particular API was available in an earlier version, and for that reason working with earlier versions of Office is not recommended.

NOTE

Office did not publish PIAs before Office 2003. Therefore, the only way to generate an interop assembly for Office 2002 or earlier versions is by importing the COM reference.

6. Close the manifest window and the assembly window.

To restore the PIA dependency

1. In **Solution Explorer**, click the **Show All Files** button. Expand the **References** folder and select `Microsoft.Office.Interop.Excel`. Press F4 to display the **Properties** window.
2. In the **Properties** window, change the **Embed Interop Types** property from **True** to **False**.

3. Repeat steps 1 and 2 in this procedure for `Microsoft.Office.Interop.Word`.
4. In C#, comment out the two calls to `Autofit` at the end of the `DisplayInExcel` method.
5. Press F5 to verify that the project still runs correctly.
6. Repeat steps 1-3 from the previous procedure to open the assembly window. Notice that `Microsoft.Office.Interop.Word` and `Microsoft.Office.Interop.Excel` are no longer in the list of embedded assemblies.
7. Double-click the **MANIFEST** icon and scroll through the list of referenced assemblies. Both `Microsoft.Office.Interop.Word` and `Microsoft.Office.Interop.Excel` are in the list. Because the application references the Excel and Word PIAs, and the **Embed Interop Types** property is set to **False**, both assemblies must exist on the end user's computer.
8. In Visual Studio, click **Clean Solution** on the **Build** menu to clean up the completed project.

See also

- [Auto-Implemented Properties \(Visual Basic\)](#)
- [Auto-Implemented Properties \(C#\)](#)
- [Collection Initializers](#)
- [Object and Collection Initializers](#)
- [Optional Parameters](#)
- [Passing Arguments by Position and by Name](#)
- [Named and Optional Arguments](#)
- [Early and Late Binding](#)
- [dynamic](#)
- [Using Type dynamic](#)
- [Lambda Expressions \(Visual Basic\)](#)
- [Lambda Expressions \(C#\)](#)
- [How to use indexed properties in COM interop programming](#)
- [Walkthrough: Embedding Type Information from Microsoft Office Assemblies in Visual Studio](#)
- [Walkthrough: Embedding Types from Managed Assemblies](#)
- [Walkthrough: Creating Your First VSTO Add-in for Excel](#)
- [COM Interop](#)
- [Interoperability](#)

Example COM Class (C# Programming Guide)

12/28/2021 • 2 minutes to read • [Edit Online](#)

The following is an example of a class that you would expose as a COM object. After this code has been placed in a .cs file and added to your project, set the **Register for COM Interop** property to **True**. For more information, see [How to: Register a Component for COM Interop](#).

Exposing Visual C# objects to COM requires declaring a class interface, an events interface if it is required, and the class itself. Class members must follow these rules to be visible to COM:

- The class must be public.
- Properties, methods, and events must be public.
- Properties and methods must be declared on the class interface.
- Events must be declared in the event interface.

Other public members in the class that are not declared in these interfaces will not be visible to COM, but they will be visible to other .NET objects.

To expose properties and methods to COM, you must declare them on the class interface and mark them with a `DispId` attribute, and implement them in the class. The order in which the members are declared in the interface is the order used for the COM vtable.

To expose events from your class, you must declare them on the events interface and mark them with a `DispId` attribute. The class should not implement this interface.

The class implements the class interface; it can implement more than one interface, but the first implementation will be the default class interface. Implement the methods and properties exposed to COM here. They must be marked public and must match the declarations in the class interface. Also, declare the events raised by the class here. They must be marked public and must match the declarations in the events interface.

Example

```
using System.Runtime.InteropServices;

namespace project_name
{
    [Guid("EAA4976A-45C3-4BC5-BC0B-E474F4C3C83F")]
    public interface ComClass1Interface
    {
    }

    [Guid("7BD20046-DF8C-44A6-8F6B-687FAA26FA71"),
        InterfaceType(ComInterfaceType.InterfaceIsIDispatch)]
    public interface ComClass1Events
    {
    }

    [Guid("0D53A3E8-E51A-49C7-944E-E72A2064F938"),
        ClassInterface(ClassInterfaceType.None),
        ComSourceInterfaces(typeof(ComClass1Events))]
    public class ComClass1 : ComClass1Interface
    {
    }
}
```

See also

- [C# Programming Guide](#)
- [Interoperability](#)
- [Build Page, Project Designer \(C#\)](#)

C# reference

12/28/2021 • 3 minutes to read • [Edit Online](#)

This section provides reference material about C# keywords, operators, special characters, preprocessor directives, compiler options, and compiler errors and warnings.

In this section

[C# Keywords](#)

Provides links to information about C# keywords and syntax.

[C# Operators](#)

Provides links to information about C# operators and syntax.

[C# Special Characters](#)

Provides links to information about special contextual characters in C# and their usage.

[C# Preprocessor Directives](#)

Provides links to information about compiler commands for embedding in C# source code.

[C# Compiler Options](#)

Includes information about compiler options and how to use them.

[C# Compiler Errors](#)

Includes code snippets that demonstrate the cause and correction of C# compiler errors and warnings.

[C# Language Specification](#)

The C# 6.0 language specification. This is a draft proposal for the C# 6.0 language. This document will be refined through work with the ECMA C# standards committee. Version 5.0 has been released in December 2017 as the [Standard ECMA-334 5th Edition](#) document.

The features that have been implemented in C# versions after 6.0 are represented in language specification proposals. These documents describe the deltas to the language spec in order to add these new features. These are in draft proposal form. These specifications will be refined and submitted to the ECMA standards committee for formal review and incorporation into a future version of the C# Standard.

[C# 7.0 Specification Proposals](#)

There are a number of new features implemented in C# 7.0. They include pattern matching, local functions, out variable declarations, throw expressions, binary literals, and digit separators. This folder contains the specifications for each of those features.

[C# 7.1 Specification Proposals](#)

There are new features added in C# 7.1. First, you can write a `Main` method that returns `Task` or `Task<int>`. This enables you to add the `async` modifier to `Main`. The `default` expression can be used without a type in locations where the type can be inferred. Also, tuple member names can be inferred. Finally, pattern matching can be used with generics.

[C# 7.2 Specification Proposals](#)

C# 7.2 added a number of small features. You can pass arguments by readonly reference using the `in` keyword. There are a number of low-level changes to support compile-time safety for `Span` and related types. You can use named arguments where later arguments are positional, in some situations. The `private protected` access modifier enables you to specify that callers are limited to derived types implemented in the same assembly. The `?:` operator can resolve to a reference to a variable. You can also format hexadecimal and binary numbers

using a leading digit separator.

[C# 7.3 Specification Proposals](#)

C# 7.3 is another point release that includes several small updates. You can use new constraints on generic type parameters. Other changes make it easier to work with `fixed` fields, including using `stackalloc` allocations. Local variables declared with the `ref` keyword may be reassigned to refer to new storage. You can place attributes on auto-implemented properties that target the compiler-generated backing field. Expression variables can be used in initializers. Tuples can be compared for equality (or inequality). There have also been some improvements to overload resolution.

[C# 8.0 Specification Proposals](#)

C# 8.0 is available with .NET Core 3.0. The features include nullable reference types, recursive pattern matching, default interface methods, async streams, ranges and indexes, pattern based using and using declarations, null coalescing assignment, and readonly instance members.

[C# 9 Specification Proposals](#)

C# 9 is available with .NET 5. The features include records, top-level statements, pattern matching enhancements, init only setters, target-typed new expressions, module initializers, extending partial methods, static anonymous functions, target-typed conditional expressions, covariant return types, extension GetEnumerator in foreach loops, lambda discard parameters, attributes on local functions, native sized integers, function pointers, suppress emitting localsinit flag, and unconstrained type parameter annotations.

[C# 10 Specification Proposals](#)

C# 10 is available with .NET 6. The features include record structs, parameterless struct constructors, global using directives, file-scoped namespaces, extended property patterns, improved interpolated strings, constant interpolated strings, lambda improvements, caller-argument expression, enhanced `#line` directives, generic attributes, improved definite assignment analysis, and `AsyncMethodBuilder` override.

Related sections

[Using the Visual Studio Development Environment for C#](#)

Provides links to conceptual and task topics that describe the IDE and Editor.

[C# Programming Guide](#)

Includes information about how to use the C# programming language.

C# language versioning

12/28/2021 • 4 minutes to read • [Edit Online](#)

The latest C# compiler determines a default language version based on your project's target framework or frameworks. Visual Studio doesn't provide a UI to change the value, but you can change it by editing the *csproj* file. The choice of default ensures that you use the latest language version compatible with your target framework. You benefit from access to the latest language features compatible with your project's target. This default choice also ensures you don't use a language that requires types or runtime behavior not available in your target framework. Choosing a language version newer than the default can cause hard to diagnose compile-time and runtime errors.

The rules in this article apply to the compiler delivered with Visual Studio 2019 or the .NET SDK. The C# compilers that are part of the Visual Studio 2017 installation or earlier .NET Core SDK versions target C# 7.0 by default.

C# 8.0 is supported only on .NET Core 3.x and newer versions. Many of the newest features require library and runtime features introduced in .NET Core 3.x:

- [Default interface implementation](#) requires new features in the .NET Core 3.0 CLR.
- [Async streams](#) require the new types [System.IAsyncDisposable](#), [System.Collections.Generic.IAsyncEnumerable<T>](#), and [System.Collections.Generic.IAsyncEnumerator<T>](#).
- [Indices and ranges](#) require the new types [System.Index](#) and [System.Range](#).
- [Nullable reference types](#) make use of several [attributes](#) to provide better warnings. Those attributes were added in .NET Core 3.0. Other target frameworks haven't been annotated with any of these attributes. That means nullable warnings may not accurately reflect potential issues.

C# 9 is supported only on .NET 5 and newer versions.

C# 10 is supported only on .NET 6 and newer versions.

Defaults

The compiler determines a default based on these rules:

TARGET FRAMEWORK	VERSION	C# LANGUAGE VERSION DEFAULT
.NET	6.x	C# 10
.NET	5.x	C# 9.0
.NET Core	3.x	C# 8.0
.NET Core	2.x	C# 7.3
.NET Standard	2.1	C# 8.0
.NET Standard	2.0	C# 7.3
.NET Standard	1.x	C# 7.3

TARGET FRAMEWORK	VERSION	C# LANGUAGE VERSION DEFAULT
.NET Framework	all	C# 7.3

When your project targets a preview framework that has a corresponding preview language version, the language version used is the preview language version. You use the latest features with that preview in any environment, without affecting projects that target a released .NET Core version.

IMPORTANT

Visual Studio 2017 added a `<LangVersion>latest</LangVersion>` entry to any project files it created. That meant C# 7.0 when it was added. However, once you upgrade to Visual Studio 2019, that means the latest released version, regardless of the target framework. These projects now [override the default behavior](#). You should edit the project file and remove that node. Then, your project will use the compiler version recommended for your target framework.

Override a default

If you must specify your C# version explicitly, you can do so in several ways:

- Manually edit your [project file](#).
- Set the language version [for multiple projects in a subdirectory](#).
- Configure the [LangVersion compiler option](#).

TIP

To know what language version you're currently using, put `#error version` (case sensitive) in your code. This makes the compiler report a compiler error, CS8304, with a message containing the compiler version being used and the current selected language version. See [#error \(C# Reference\)](#) for more information.

Edit the project file

You can set the language version in your project file. For example, if you explicitly want access to preview features, add an element like this:

```
<PropertyGroup>
  <LangVersion>preview</LangVersion>
</PropertyGroup>
```

The value `preview` uses the latest available preview C# language version that your compiler supports.

Configure multiple projects

To configure multiple projects, you can create a **Directory.Build.props** file that contains the `<LangVersion>` element. You typically do that in your solution directory. Add the following to a **Directory.Build.props** file in your solution directory:

```
<Project>
  <PropertyGroup>
    <LangVersion>preview</LangVersion>
  </PropertyGroup>
</Project>
```

Builds in all subdirectories of the directory containing that file will use the preview C# version. For more information, see [Customize your build](#).

C# language version reference

The following table shows all current C# language versions. Your compiler may not necessarily understand every value if it's older. If you install the latest .NET SDK, then you have access to everything listed.

VALUE	MEANING
<code>preview</code>	The compiler accepts all valid language syntax from the latest preview version.
<code>latest</code>	The compiler accepts syntax from the latest released version of the compiler (including minor version).
<code>latestMajor</code> (<code>default</code>)	The compiler accepts syntax from the latest released major version of the compiler.
<code>10.0</code>	The compiler accepts only syntax that is included in C# 10 or lower.
<code>9.0</code>	The compiler accepts only syntax that is included in C# 9 or lower.
<code>8.0</code>	The compiler accepts only syntax that is included in C# 8.0 or lower.
<code>7.3</code>	The compiler accepts only syntax that is included in C# 7.3 or lower.
<code>7.2</code>	The compiler accepts only syntax that is included in C# 7.2 or lower.
<code>7.1</code>	The compiler accepts only syntax that is included in C# 7.1 or lower.
<code>7</code>	The compiler accepts only syntax that is included in C# 7.0 or lower.
<code>6</code>	The compiler accepts only syntax that is included in C# 6.0 or lower.
<code>5</code>	The compiler accepts only syntax that is included in C# 5.0 or lower.
<code>4</code>	The compiler accepts only syntax that is included in C# 4.0 or lower.
<code>3</code>	The compiler accepts only syntax that is included in C# 3.0 or lower.
<code>ISO-2</code> (or <code>2</code>)	The compiler accepts only syntax that is included in ISO/IEC 23270:2006 C# (2.0).
<code>ISO-1</code> (or <code>1</code>)	The compiler accepts only syntax that is included in ISO/IEC 23270:2003 C# (1.0/1.2).

Value types (C# reference)

12/28/2021 • 3 minutes to read • [Edit Online](#)

Value types and *reference types* are the two main categories of C# types. A variable of a value type contains an instance of the type. This differs from a variable of a reference type, which contains a reference to an instance of the type. By default, on [assignment](#), passing an argument to a method, and returning a method result, variable values are copied. In the case of value-type variables, the corresponding type instances are copied. The following example demonstrates that behavior:

```
using System;

public struct MutablePoint
{
    public int X;
    public int Y;

    public MutablePoint(int x, int y) => (X, Y) = (x, y);

    public override string ToString() => $"({X}, {Y})";
}

public class Program
{
    public static void Main()
    {
        var p1 = new MutablePoint(1, 2);
        var p2 = p1;
        p2.Y = 200;
        Console.WriteLine($"{nameof(p1)} after {nameof(p2)} is modified: {p1}");
        Console.WriteLine($"{nameof(p2)}: {p2}");

        MutateAndDisplay(p2);
        Console.WriteLine($"{nameof(p2)} after passing to a method: {p2}");
    }

    private static void MutateAndDisplay(MutablePoint p)
    {
        p.X = 100;
        Console.WriteLine($"Point mutated in a method: {p}");
    }
}

// Expected output:
// p1 after p2 is modified: (1, 2)
// p2: (1, 200)
// Point mutated in a method: (100, 200)
// p2 after passing to a method: (1, 200)
```

As the preceding example shows, operations on a value-type variable affect only that instance of the value type, stored in the variable.

If a value type contains a data member of a reference type, only the reference to the instance of the reference type is copied when a value-type instance is copied. Both the copy and original value-type instance have access to the same reference-type instance. The following example demonstrates that behavior:

```

using System;
using System.Collections.Generic;

public struct TaggedInteger
{
    public int Number;
    private List<string> tags;

    public TaggedInteger(int n)
    {
        Number = n;
        tags = new List<string>();
    }

    public void AddTag(string tag) => tags.Add(tag);

    public override string ToString() => $"{Number} [{string.Join(", ", tags)}]";
}

public class Program
{
    public static void Main()
    {
        var n1 = new TaggedInteger(0);
        n1.AddTag("A");
        Console.WriteLine(n1); // output: 0 [A]

        var n2 = n1;
        n2.Number = 7;
        n2.AddTag("B");

        Console.WriteLine(n1); // output: 0 [A, B]
        Console.WriteLine(n2); // output: 7 [A, B]
    }
}

```

NOTE

To make your code less error-prone and more robust, define and use immutable value types. This article uses mutable value types only for demonstration purposes.

Kinds of value types and type constraints

A value type can be one of the two following kinds:

- a [structure type](#), which encapsulates data and related functionality
- an [enumeration type](#), which is defined by a set of named constants and represents a choice or a combination of choices

A [nullable value type](#) `T?` represents all values of its underlying value type `T` and an additional [null](#) value. You cannot assign `null` to a variable of a value type, unless it's a nullable value type.

You can use the [struct constraint](#) to specify that a type parameter is a non-nullable value type. Both structure and enumeration types satisfy the `struct` constraint. Beginning with C# 7.3, you can use `System.Enum` in a base class constraint (that is known as the [enum constraint](#)) to specify that a type parameter is an enumeration type.

Built-in value types

C# provides the following built-in value types, also known as *simple types*.

- [Integral numeric types](#)
- [Floating-point numeric types](#)
- `bool` that represents a Boolean value
- `char` that represents a Unicode UTF-16 character

All simple types are structure types and differ from other structure types in that they permit certain additional operations:

- You can use literals to provide a value of a simple type. For example, `'A'` is a literal of the type `char` and `2001` is a literal of the type `int`.
- You can declare constants of the simple types with the `const` keyword. It's not possible to have constants of other structure types.
- Constant expressions, whose operands are all constants of the simple types, are evaluated at compile time.

Beginning with C# 7.0, C# supports [value tuples](#). A value tuple is a value type, but not a simple type.

C# language specification

For more information, see the following sections of the [C# language specification](#):

- [Value types](#)
- [Simple types](#)
- [Variables](#)

See also

- [C# reference](#)
- [System.ValueType](#)
- [Reference types](#)

Integral numeric types (C# reference)

12/28/2021 • 4 minutes to read • [Edit Online](#)

The *integral numeric types* represent integer numbers. All integral numeric types are [value types](#). They are also [simple types](#) and can be initialized with [literals](#). All integral numeric types support [arithmetic](#), [bitwise logical](#), [comparison](#), and [equality](#) operators.

Characteristics of the integral types

C# supports the following predefined integral types:

C# TYPE/KEYWORD	RANGE	SIZE	.NET TYPE
<code>sbyte</code>	-128 to 127	Signed 8-bit integer	System.SByte
<code>byte</code>	0 to 255	Unsigned 8-bit integer	System.Byte
<code>short</code>	-32,768 to 32,767	Signed 16-bit integer	System.Int16
<code>ushort</code>	0 to 65,535	Unsigned 16-bit integer	System.UInt16
<code>int</code>	-2,147,483,648 to 2,147,483,647	Signed 32-bit integer	System.Int32
<code>uint</code>	0 to 4,294,967,295	Unsigned 32-bit integer	System.UInt32
<code>long</code>	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807	Signed 64-bit integer	System.Int64
<code>ulong</code>	0 to 18,446,744,073,709,551,615	Unsigned 64-bit integer	System.UInt64
<code>nint</code>	Depends on platform	Signed 32-bit or 64-bit integer	System.IntPtr
<code>nuint</code>	Depends on platform	Unsigned 32-bit or 64-bit integer	System.UIntPtr

In all of the table rows except the last two, each C# type keyword from the leftmost column is an alias for the corresponding .NET type. The keyword and .NET type name are interchangeable. For example, the following declarations declare variables of the same type:

```
int a = 123;  
System.Int32 b = 123;
```

The `nint` and `nuint` types in the last two rows of the table are native-sized integers. They are represented internally by the indicated .NET types, but in each case the keyword and the .NET type are not interchangeable.

The compiler provides operations and conversions for `nint` and `nuint` as integer types that it doesn't provide for the pointer types `System.IntPtr` and `System.UIntPtr`. For more information, see [nint and nuint types](#).

The default value of each integral type is zero, `0`. Each of the integral types except the native-sized types has `MinValue` and `MaxValue` constants that provide the minimum and maximum value of that type.

Use the [System.Numerics.BigInteger](#) structure to represent a signed integer with no upper or lower bounds.

Integer literals

Integer literals can be

- *decimal*: without any prefix
- *hexadecimal*: with the `0x` or `0X` prefix
- *binary*: with the `0b` or `0B` prefix (available in C# 7.0 and later)

The following code demonstrates an example of each:

```
var decimalLiteral = 42;
var hexLiteral = 0x2A;
var binaryLiteral = 0b_0010_1010;
```

The preceding example also shows the use of `_` as a *digit separator*, which is supported starting with C# 7.0. You can use the digit separator with all kinds of numeric literals.

The type of an integer literal is determined by its suffix as follows:

- If the literal has no suffix, its type is the first of the following types in which its value can be represented:

`int`, `uint`, `long`, `ulong`.

NOTE

Literals are interpreted as positive values. For example, the literal `0xFF_FF_FF_FF` represents the number `4294967295` of the `uint` type, though it has the same bit representation as the number `-1` of the `int` type. If you need a value of a certain type, cast a literal to that type. Use the `unchecked` operator, if a literal value cannot be represented in the target type. For example, `unchecked((int)0xFF_FF_FF_FF)` produces `-1`.

- If the literal is suffixed by `U` or `u`, its type is the first of the following types in which its value can be represented: `uint`, `ulong`.
- If the literal is suffixed by `L` or `l`, its type is the first of the following types in which its value can be represented: `long`, `ulong`.

NOTE

You can use the lowercase letter `l` as a suffix. However, this generates a compiler warning because the letter `l` can be confused with the digit `1`. Use `L` for clarity.

- If the literal is suffixed by `UL`, `Ul`, `uL`, `ul`, `LU`, `Lu`, `lU`, or `lu`, its type is `ulong`.

If the value represented by an integer literal exceeds `UInt64.MaxValue`, a compiler error [CS1021](#) occurs.

If the determined type of an integer literal is `int` and the value represented by the literal is within the range of the destination type, the value can be implicitly converted to `sbyte`, `byte`, `short`, `ushort`, `uint`, `ulong`, `nint` or `nuint`:

```
byte a = 17;
byte b = 300;    // CS0031: Constant value '300' cannot be converted to a 'byte'
```

As the preceding example shows, if the literal's value is not within the range of the destination type, a compiler error [CS0031](#) occurs.

You can also use a cast to convert the value represented by an integer literal to the type other than the determined type of the literal:

```
var signedByte = (sbyte)42;
var longVariable = (long)42;
```

Conversions

You can convert any integral numeric type to any other integral numeric type. If the destination type can store all values of the source type, the conversion is implicit. Otherwise, you need to use a [cast expression](#) to perform an explicit conversion. For more information, see [Built-in numeric conversions](#).

C# language specification

For more information, see the following sections of the [C# language specification](#):

- [Integral types](#)
- [Integer literals](#)

See also

- [C# reference](#)
- [Value types](#)
- [Floating-point types](#)
- [Standard numeric format strings](#)
- [Numerics in .NET](#)

nint and nuint types (C# reference)

12/28/2021 • 2 minutes to read • [Edit Online](#)

Starting in C# 9.0, you can use the `nint` and `nuint` keywords to define *native-sized integers*. These are 32-bit integers when running in a 32-bit process, or 64-bit integers when running in a 64-bit process. They can be used for interop scenarios, low-level libraries, and to optimize performance in scenarios where integer math is used extensively.

The native-sized integer types are represented internally as the .NET types `System.IntPtr` and `System.UIntPtr`. Unlike other numeric types, the keywords are not simply aliases for the types. The following statements are not equivalent:

```
nint a = 1;
System.IntPtr a = 1;
```

The compiler provides operations and conversions for `nint` and `nuint` that are appropriate for integer types.

Run-time native integer size

To get the size of a native-sized integer at run time, you can use `sizeof()`. However, the code must be compiled in an unsafe context. For example:

```
Console.WriteLine($"size of nint = {sizeof(nint)}");
Console.WriteLine($"size of nuint = {sizeof(nuint)}");

// output when run in a 64-bit process
//size of nint = 8
//size of nuint = 8

// output when run in a 32-bit process
//size of nint = 4
//size of nuint = 4
```

You can also get the equivalent value from the static `IntPtr.Size` and `UIntPtr.Size` properties.

MinValue and MaxValue

To get the minimum and maximum values of native-sized integers at run time, use `MinValue` and `MaxValue` as static properties with the `nint` and `nuint` keywords, as in the following example:

```

Console.WriteLine($"nint.MinValue = {nint.MinValue}");
Console.WriteLine($"nint.MaxValue = {nint.MaxValue}");
Console.WriteLine($"nuint.MinValue = {nuint.MinValue}");
Console.WriteLine($"nuint.MaxValue = {nuint.MaxValue}");

// output when run in a 64-bit process
//nint.MinValue = -9223372036854775808
//nint.MaxValue = 9223372036854775807
//nuint.MinValue = 0
//nuint.MaxValue = 18446744073709551615

// output when run in a 32-bit process
//nint.MinValue = -2147483648
//nint.MaxValue = 2147483647
//nuint.MinValue = 0
//nuint.MaxValue = 4294967295

```

Constants

You can use constant values in the following ranges:

- For `nint`: [Int32.MinValue](#) to [Int32.MaxValue](#).
- For `nuint`: [UInt32.MinValue](#) to [UInt32.MaxValue](#).

Conversions

The compiler provides implicit and explicit conversions to other numeric types. For more information, see [Built-in numeric conversions](#).

Literals

There's no direct syntax for native-sized integer literals. There's no suffix to indicate that a literal is a native-sized integer, such as `L` to indicate a `long`. You can use implicit or explicit casts of other integer values instead. For example:

```

nint a = 42
nint a = (nint)42;

```

Unsupported IntPtr/UIntPtr members

The following members of [IntPtr](#) and [UIntPtr](#) aren't supported for `nint` and `nuint` types:

- Parameterized constructors
- [Add\(IntPtr, Int32\)](#)
- [CompareTo](#)
- [Size](#) - Use [sizeof\(\)](#) instead. Although `nint.Size` isn't supported, you can use `IntPtr.Size` to get an equivalent value.
- [Subtract\(IntPtr, Int32\)](#)
- [ToInt32](#)
- [ToInt64](#)
- [ToPointer](#)
- [Zero](#) - Use 0 instead.

C# language specification

For more information, see the [C# language specification](#) and the [Native-sized integers](#) section of the C# 9.0 feature proposal notes.

See also

- [C# reference](#)
- [Value types](#)
- [Integral numeric types](#)
- [Built-in numeric conversions](#)

Floating-point numeric types (C# reference)

12/28/2021 • 4 minutes to read • [Edit Online](#)

The *floating-point numeric types* represent real numbers. All floating-point numeric types are [value types](#). They are also [simple types](#) and can be initialized with [literals](#). All floating-point numeric types support [arithmetic](#), [comparison](#), and [equality](#) operators.

Characteristics of the floating-point types

C# supports the following predefined floating-point types:

C# TYPE/KEYWORD	APPROXIMATE RANGE	PRECISION	SIZE	.NET TYPE
<code>float</code>	$\pm 1.5 \times 10^{-45}$ to $\pm 3.4 \times 10^{38}$	~6-9 digits	4 bytes	System.Single
<code>double</code>	$\pm 5.0 \times 10^{-324}$ to $\pm 1.7 \times 10^{308}$	~15-17 digits	8 bytes	System.Double
<code>decimal</code>	$\pm 1.0 \times 10^{-28}$ to $\pm 7.9228 \times 10^{28}$	28-29 digits	16 bytes	System.Decimal

In the preceding table, each C# type keyword from the leftmost column is an alias for the corresponding .NET type. They are interchangeable. For example, the following declarations declare variables of the same type:

```
double a = 12.3;
System.Double b = 12.3;
```

The default value of each floating-point type is zero, `0`. Each of the floating-point types has the `MinValue` and `MaxValue` constants that provide the minimum and maximum finite value of that type. The `float` and `double` types also provide constants that represent not-a-number and infinity values. For example, the `double` type provides the following constants: [Double.NaN](#), [Double.NegativeInfinity](#), and [Double.PositiveInfinity](#).

The `decimal` type is appropriate when the required degree of precision is determined by the number of digits to the right of the decimal point. Such numbers are commonly used in financial applications, for currency amounts (for example, \$1.00), interest rates (for example, 2.625%), and so forth. Even numbers that are precise to only one decimal digit are handled more accurately by the `decimal` type: 0.1, for example, can be exactly represented by a `decimal` instance, while there's no `double` or `float` instance that exactly represents 0.1. Because of this difference in numeric types, unexpected rounding errors can occur in arithmetic calculations when you use `double` or `float` for decimal data. You can use `double` instead of `decimal` when optimizing performance is more important than ensuring accuracy. However, any difference in performance would go unnoticed by all but the most calculation-intensive applications. Another possible reason to avoid `decimal` is to minimize storage requirements. For example, [ML.NET](#) uses `float` because the difference between 4 bytes and 16 bytes adds up for very large data sets. For more information, see [System.Decimal](#).

You can mix [integral](#) types and the `float` and `double` types in an expression. In this case, integral types are implicitly converted to one of the floating-point types and, if necessary, the `float` type is implicitly converted to `double`. The expression is evaluated as follows:

- If there is `double` type in the expression, the expression evaluates to `double`, or to `bool` in relational and equality comparisons.
- If there is no `double` type in the expression, the expression evaluates to `float`, or to `bool` in relational and equality comparisons.

You can also mix integral types and the `decimal` type in an expression. In this case, integral types are implicitly converted to the `decimal` type and the expression evaluates to `decimal`, or to `bool` in relational and equality comparisons.

You cannot mix the `decimal` type with the `float` and `double` types in an expression. In this case, if you want to perform arithmetic, comparison, or equality operations, you must explicitly convert the operands either from or to the `decimal` type, as the following example shows:

```
double a = 1.0;
decimal b = 2.1m;
Console.WriteLine(a + (double)b);
Console.WriteLine((decimal)a + b);
```

You can use either [standard numeric format strings](#) or [custom numeric format strings](#) to format a floating-point value.

Real literals

The type of a real literal is determined by its suffix as follows:

- The literal without suffix or with the `d` or `D` suffix is of type `double`
- The literal with the `f` or `F` suffix is of type `float`
- The literal with the `m` or `M` suffix is of type `decimal`

The following code demonstrates an example of each:

```
double d = 3D;
d = 4d;
d = 3.934_001;

float f = 3_000.5F;
f = 5.4f;

decimal myMoney = 3_000.5m;
myMoney = 400.75M;
```

The preceding example also shows the use of `_` as a *digit separator*, which is supported starting with C# 7.0. You can use the digit separator with all kinds of numeric literals.

You can also use scientific notation, that is, specify an exponent part of a real literal, as the following example shows:

```
double d = 0.42e2;
Console.WriteLine(d); // output 42

float f = 134.45E-2f;
Console.WriteLine(f); // output: 1.3445

decimal m = 1.5E6m;
Console.WriteLine(m); // output: 1500000
```

Conversions

There is only one implicit conversion between floating-point numeric types: from `float` to `double`. However, you can convert any floating-point type to any other floating-point type with the [explicit cast](#). For more information, see [Built-in numeric conversions](#).

C# language specification

For more information, see the following sections of the [C# language specification](#):

- [Floating-point types](#)
- [The decimal type](#)
- [Real literals](#)

See also

- [C# reference](#)
- [Value types](#)
- [Integral types](#)
- [Standard numeric format strings](#)
- [Numerics in .NET](#)
- [System.Numerics.Complex](#)

Built-in numeric conversions (C# reference)

12/28/2021 • 4 minutes to read • [Edit Online](#)

C# provides a set of [integral](#) and [floating-point](#) numeric types. There exists a conversion between any two numeric types, either implicit or explicit. You must use a [cast expression](#) to perform an explicit conversion.

Implicit numeric conversions

The following table shows the predefined implicit conversions between the built-in numeric types:

FROM	TO
sbyte	short , int , long , float , double , decimal , OR nint
byte	short , ushort , int , uint , long , ulong , float , double , decimal , nint , OR nuint
short	int , long , float , double , OR decimal , OR nint
ushort	int , uint , long , ulong , float , double , OR decimal , nint , OR nuint
int	long , float , double , OR decimal , nint
uint	long , ulong , float , double , OR decimal , OR nuint
long	float , double , OR decimal
ulong	float , double , OR decimal
float	double
nint	long , float , double , OR decimal
nuint	ulong , float , double , OR decimal

NOTE

The implicit conversions from [int](#), [uint](#), [long](#), [ulong](#), [nint](#), OR [nuint](#) to [float](#) and from [long](#), [ulong](#), [nint](#), OR [nuint](#) to [double](#) may cause a loss of precision, but never a loss of an order of magnitude. The other implicit numeric conversions never lose any information.

Also note that

- Any [integral numeric type](#) is implicitly convertible to any [floating-point numeric type](#).
- There are no implicit conversions to the [byte](#) and [sbyte](#) types. There are no implicit conversions from the [double](#) and [decimal](#) types.

- There are no implicit conversions between the `decimal` type and the `float` or `double` types.
- A value of a constant expression of type `int` (for example, a value represented by an integer literal) can be implicitly converted to `sbyte`, `byte`, `short`, `ushort`, `uint`, `ulong`, `nint`, or `nuint`, if it's within the range of the destination type:

```
byte a = 13;
byte b = 300; // CS0031: Constant value '300' cannot be converted to a 'byte'
```

As the preceding example shows, if the constant value is not within the range of the destination type, a compiler error [CS0031](#) occurs.

Explicit numeric conversions

The following table shows the predefined explicit conversions between the built-in numeric types for which there is no [implicit conversion](#):

FROM	TO
sbyte	<code>byte</code> , <code>ushort</code> , <code>uint</code> , or <code>ulong</code> , or <code>nuint</code>
byte	<code>sbyte</code>
short	<code>sbyte</code> , <code>byte</code> , <code>ushort</code> , <code>uint</code> , <code>ulong</code> , or <code>nuint</code>
ushort	<code>sbyte</code> , <code>byte</code> , or <code>short</code>
int	<code>sbyte</code> , <code>byte</code> , <code>short</code> , <code>ushort</code> , <code>uint</code> , <code>ulong</code> , or <code>nuint</code>
uint	<code>sbyte</code> , <code>byte</code> , <code>short</code> , <code>ushort</code> , or <code>int</code>
long	<code>sbyte</code> , <code>byte</code> , <code>short</code> , <code>ushort</code> , <code>int</code> , <code>uint</code> , <code>ulong</code> , <code>nint</code> , or <code>nuint</code>
ulong	<code>sbyte</code> , <code>byte</code> , <code>short</code> , <code>ushort</code> , <code>int</code> , <code>uint</code> , <code>long</code> , <code>nint</code> , or <code>nuint</code>
float	<code>sbyte</code> , <code>byte</code> , <code>short</code> , <code>ushort</code> , <code>int</code> , <code>uint</code> , <code>long</code> , <code>ulong</code> , <code>decimal</code> , <code>nint</code> , or <code>nuint</code>
double	<code>sbyte</code> , <code>byte</code> , <code>short</code> , <code>ushort</code> , <code>int</code> , <code>uint</code> , <code>long</code> , <code>ulong</code> , <code>float</code> , <code>decimal</code> , <code>nint</code> , or <code>nuint</code>
decimal	<code>sbyte</code> , <code>byte</code> , <code>short</code> , <code>ushort</code> , <code>int</code> , <code>uint</code> , <code>long</code> , <code>ulong</code> , <code>float</code> , <code>double</code> , <code>nint</code> , or <code>nuint</code>
nint	<code>sbyte</code> , <code>byte</code> , <code>short</code> , <code>ushort</code> , <code>int</code> , <code>uint</code> , <code>ulong</code> , or <code>nuint</code>
nuint	<code>sbyte</code> , <code>byte</code> , <code>short</code> , <code>ushort</code> , <code>int</code> , <code>uint</code> , <code>long</code> , or <code>nint</code>

NOTE

An explicit numeric conversion might result in data loss or throw an exception, typically an [OverflowException](#).

Also note that

- When you convert a value of an integral type to another integral type, the result depends on the overflow [checking context](#). In a checked context, the conversion succeeds if the source value is within the range of the destination type. Otherwise, an [OverflowException](#) is thrown. In an unchecked context, the conversion always succeeds, and proceeds as follows:
 - If the source type is larger than the destination type, then the source value is truncated by discarding its "extra" most significant bits. The result is then treated as a value of the destination type.
 - If the source type is smaller than the destination type, then the source value is either sign-extended or zero-extended so that it's of the same size as the destination type. Sign-extension is used if the source type is signed; zero-extension is used if the source type is unsigned. The result is then treated as a value of the destination type.
 - If the source type is the same size as the destination type, then the source value is treated as a value of the destination type.
- When you convert a `decimal` value to an integral type, this value is rounded towards zero to the nearest integral value. If the resulting integral value is outside the range of the destination type, an [OverflowException](#) is thrown.
- When you convert a `double` or `float` value to an integral type, this value is rounded towards zero to the nearest integral value. If the resulting integral value is outside the range of the destination type, the result depends on the overflow [checking context](#). In a checked context, an [OverflowException](#) is thrown, while in an unchecked context, the result is an unspecified value of the destination type.
- When you convert `double` to `float`, the `double` value is rounded to the nearest `float` value. If the `double` value is too small or too large to fit into the `float` type, the result is zero or infinity.
- When you convert `float` or `double` to `decimal`, the source value is converted to `decimal` representation and rounded to the nearest number after the 28th decimal place if required. Depending on the value of the source value, one of the following results may occur:
 - If the source value is too small to be represented as a `decimal`, the result becomes zero.
 - If the source value is NaN (not a number), infinity, or too large to be represented as a `decimal`, an [OverflowException](#) is thrown.
- When you convert `decimal` to `float` or `double`, the source value is rounded to the nearest `float` or `double` value, respectively.

C# language specification

For more information, see the following sections of the [C# language specification](#):

- [Implicit numeric conversions](#)
- [Explicit numeric conversions](#)

See also

- [C# reference](#)
- [Casting and type conversions](#)

bool (C# reference)

12/28/2021 • 2 minutes to read • [Edit Online](#)

The `bool` type keyword is an alias for the .NET [System.Boolean](#) structure type that represents a Boolean value, which can be either `true` or `false`.

To perform logical operations with values of the `bool` type, use [Boolean logical](#) operators. The `bool` type is the result type of [comparison](#) and [equality](#) operators. A `bool` expression can be a controlling conditional expression in the [if](#), [do](#), [while](#), and [for](#) statements and in the [conditional operator](#) `?:`.

The default value of the `bool` type is `false`.

Literals

You can use the `true` and `false` literals to initialize a `bool` variable or to pass a `bool` value:

```
bool check = true;
Console.WriteLine(check ? "Checked" : "Not checked"); // output: Checked

Console.WriteLine(false ? "Checked" : "Not checked"); // output: Not checked
```

Three-valued Boolean logic

Use the nullable `bool?` type, if you need to support the three-valued logic, for example, when you work with databases that support a three-valued Boolean type. For the `bool?` operands, the predefined `&` and `|` operators support the three-valued logic. For more information, see the [Nullable Boolean logical operators](#) section of the [Boolean logical operators](#) article.

For more information about nullable value types, see [Nullable value types](#).

Conversions

C# provides only two conversions that involve the `bool` type. Those are an implicit conversion to the corresponding nullable `bool?` type and an explicit conversion from the `bool?` type. However, .NET provides additional methods that you can use to convert to or from the `bool` type. For more information, see the [Converting to and from Boolean values](#) section of the [System.Boolean](#) API reference page.

C# language specification

For more information, see [The bool type](#) section of the [C# language specification](#).

See also

- [C# reference](#)
- [Value types](#)
- [true and false operators](#)

char (C# reference)

12/28/2021 • 2 minutes to read • [Edit Online](#)

The `char` type keyword is an alias for the .NET [System.Char](#) structure type that represents a Unicode UTF-16 character.

TYPE	RANGE	SIZE	.NET TYPE
<code>char</code>	U+0000 to U+FFFF	16 bit	System.Char

The default value of the `char` type is `\0`, that is, U+0000.

The `char` type supports [comparison](#), [equality](#), [increment](#), and [decrement](#) operators. Moreover, for `char` operands, [arithmetic](#) and [bitwise logical](#) operators perform an operation on the corresponding character codes and produce the result of the `int` type.

The [string](#) type represents text as a sequence of `char` values.

Literals

You can specify a `char` value with:

- a character literal.
- a Unicode escape sequence, which is `\u` followed by the four-symbol hexadecimal representation of a character code.
- a hexadecimal escape sequence, which is `\x` followed by the hexadecimal representation of a character code.

```
var chars = new[]
{
    'j',
    '\u006A',
    '\x006A',
    (char)106,
};
Console.WriteLine(string.Join(" ", chars)); // output: j j j j
```

As the preceding example shows, you can also cast the value of a character code into the corresponding `char` value.

NOTE

In the case of a Unicode escape sequence, you must specify all four hexadecimal digits. That is, `\u006A` is a valid escape sequence, while `\u06A` and `\u6A` are not valid.

In the case of a hexadecimal escape sequence, you can omit the leading zeros. That is, the `\x006A`, `\x06A`, and `\x6A` escape sequences are valid and correspond to the same character.

Conversions

The `char` type is implicitly convertible to the following [integral](#) types: `ushort`, `int`, `uint`, `long`, and `ulong`.

It's also implicitly convertible to the built-in [floating-point](#) numeric types: `float`, `double`, and `decimal`. It's explicitly convertible to `sbyte`, `byte`, and `short` integral types.

There are no implicit conversions from other types to the `char` type. However, any [integral](#) or [floating-point](#) numeric type is explicitly convertible to `char`.

C# language specification

For more information, see the [Integral types](#) section of the [C# language specification](#).

See also

- [C# reference](#)
- [Value types](#)
- [Strings](#)
- [System.Text.Rune](#)
- [Character encoding in .NET](#)

Enumeration types (C# reference)

12/28/2021 • 3 minutes to read • [Edit Online](#)

An *enumeration type* (or *enum type*) is a [value type](#) defined by a set of named constants of the underlying [integral numeric](#) type. To define an enumeration type, use the `enum` keyword and specify the names of *enum members*.

```
enum Season
{
    Spring,
    Summer,
    Autumn,
    Winter
}
```

By default, the associated constant values of enum members are of type `int`; they start with zero and increase by one following the definition text order. You can explicitly specify any other [integral numeric](#) type as an underlying type of an enumeration type. You can also explicitly specify the associated constant values, as the following example shows:

```
enum ErrorCode : ushort
{
    None = 0,
    Unknown = 1,
    ConnectionLost = 100,
    OutlierReading = 200
}
```

You cannot define a method inside the definition of an enumeration type. To add functionality to an enumeration type, create an [extension method](#).

The default value of an enumeration type `E` is the value produced by expression `(E)0`, even if zero doesn't have the corresponding enum member.

You use an enumeration type to represent a choice from a set of mutually exclusive values or a combination of choices. To represent a combination of choices, define an enumeration type as bit flags.

Enumeration types as bit flags

If you want an enumeration type to represent a combination of choices, define enum members for those choices such that an individual choice is a bit field. That is, the associated values of those enum members should be the powers of two. Then, you can use the [bitwise logical operators](#) `|` or `&` to combine choices or intersect combinations of choices, respectively. To indicate that an enumeration type declares bit fields, apply the [Flags](#) attribute to it. As the following example shows, you can also include some typical combinations in the definition of an enumeration type.


```

[Flags]
public enum Days
{
    None      = 0b_0000_0000, // 0
    Monday    = 0b_0000_0001, // 1
    Tuesday   = 0b_0000_0010, // 2
    Wednesday = 0b_0000_0100, // 4
    Thursday  = 0b_0000_1000, // 8
    Friday    = 0b_0001_0000, // 16
    Saturday  = 0b_0010_0000, // 32
    Sunday    = 0b_0100_0000, // 64
    Weekend   = Saturday | Sunday
}

public class FlagsEnumExample
{
    public static void Main()
    {
        Days meetingDays = Days.Monday | Days.Wednesday | Days.Friday;
        Console.WriteLine(meetingDays);
        // Output:
        // Monday, Wednesday, Friday

        Days workingFromHomeDays = Days.Thursday | Days.Friday;
        Console.WriteLine($"Join a meeting by phone on {meetingDays & workingFromHomeDays}");
        // Output:
        // Join a meeting by phone on Friday

        bool isMeetingOnTuesday = (meetingDays & Days.Tuesday) == Days.Tuesday;
        Console.WriteLine($"Is there a meeting on Tuesday: {isMeetingOnTuesday}");
        // Output:
        // Is there a meeting on Tuesday: False

        var a = (Days)37;
        Console.WriteLine(a);
        // Output:
        // Monday, Wednesday, Saturday
    }
}

```

For more information and examples, see the [System.FlagsAttribute](#) API reference page and the [Non-exclusive members and the Flags attribute](#) section of the [System.Enum](#) API reference page.

The System.Enum type and enum constraint

The [System.Enum](#) type is the abstract base class of all enumeration types. It provides a number of methods to get information about an enumeration type and its values. For more information and examples, see the [System.Enum](#) API reference page.

Beginning with C# 7.3, you can use `System.Enum` in a base class constraint (that is known as the [enum constraint](#)) to specify that a type parameter is an enumeration type. Any enumeration type also satisfies the `struct` constraint, which is used to specify that a type parameter is a non-nullable value type.

Conversions

For any enumeration type, there exist explicit conversions between the enumeration type and its underlying integral type. If you [cast](#) an enum value to its underlying type, the result is the associated integral value of an enum member.

```

public enum Season
{
    Spring,
    Summer,
    Autumn,
    Winter
}

public class EnumConversionExample
{
    public static void Main()
    {
        Season a = Season.Autumn;
        Console.WriteLine($"Integral value of {a} is {(int)a}"); // output: Integral value of Autumn is 2

        var b = (Season)1;
        Console.WriteLine(b); // output: Summer

        var c = (Season)4;
        Console.WriteLine(c); // output: 4
    }
}

```

Use the [Enum.IsDefined](#) method to determine whether an enumeration type contains an enum member with the certain associated value.

For any enumeration type, there exist [boxing and unboxing](#) conversions to and from the [System.Enum](#) type, respectively.

C# language specification

For more information, see the following sections of the [C# language specification](#):

- [Enums](#)
- [Enum values and operations](#)
- [Enumeration logical operators](#)
- [Enumeration comparison operators](#)
- [Explicit enumeration conversions](#)
- [Implicit enumeration conversions](#)

See also

- [C# reference](#)
- [Enumeration format strings](#)
- [Design guidelines - Enum design](#)
- [Design guidelines - Enum naming conventions](#)
- [switch expression](#)
- [switch statement](#)

Structure types (C# reference)

12/28/2021 • 11 minutes to read • [Edit Online](#)

A *structure type* (or *struct type*) is a [value type](#) that can encapsulate data and related functionality. You use the `struct` keyword to define a structure type:

```
public struct Coords
{
    public Coords(double x, double y)
    {
        X = x;
        Y = y;
    }

    public double X { get; }
    public double Y { get; }

    public override string ToString() => $"({X}, {Y})";
}
```

Structure types have *value semantics*. That is, a variable of a structure type contains an instance of the type. By default, variable values are copied on assignment, passing an argument to a method, and returning a method result. In the case of a structure-type variable, an instance of the type is copied. For more information, see [Value types](#).

Typically, you use structure types to design small data-centric types that provide little or no behavior. For example, .NET uses structure types to represent a number (both [integer](#) and [real](#)), a [Boolean value](#), a [Unicode character](#), a [time instance](#). If you're focused on the behavior of a type, consider defining a [class](#). Class types have *reference semantics*. That is, a variable of a class type contains a reference to an instance of the type, not the instance itself.

Because structure types have value semantics, we recommend you to define *immutable* structure types.

`readonly` `struct`

Beginning with C# 7.2, you use the `readonly` modifier to declare that a structure type is immutable. All data members of a `readonly` struct must be read-only as follows:

- Any field declaration must have the `readonly` modifier
- Any property, including auto-implemented ones, must be read-only. In C# 9.0 and later, a property may have an `init` accessor.

That guarantees that no member of a `readonly` struct modifies the state of the struct. In C# 8.0 and later, that means that other instance members except constructors are implicitly `readonly`.

NOTE

In a `readonly` struct, a data member of a mutable reference type still can mutate its own state. For example, you can't replace a `List<T>` instance, but you can add new elements to it.

The following code defines a `readonly` struct with init-only property setters, available in C# 9.0 and later:

```
public readonly struct Coords
{
    public Coords(double x, double y)
    {
        X = x;
        Y = y;
    }

    public double X { get; init; }
    public double Y { get; init; }

    public override string ToString() => $"({X}, {Y})";
}
```

readonly instance members

Beginning with C# 8.0, you can also use the `readonly` modifier to declare that an instance member doesn't modify the state of a struct. If you can't declare the whole structure type as `readonly`, use the `readonly` modifier to mark the instance members that don't modify the state of the struct.

Within a `readonly` instance member, you can't assign to structure's instance fields. However, a `readonly` member can call a non-`readonly` member. In that case the compiler creates a copy of the structure instance and calls the non-`readonly` member on that copy. As a result, the original structure instance is not modified.

Typically, you apply the `readonly` modifier to the following kinds of instance members:

- methods:

```
public readonly double Sum()
{
    return X + Y;
}
```

You can also apply the `readonly` modifier to methods that override methods declared in [System.Object](#):

```
public readonly override string ToString() => $"({X}, {Y})";
```

- properties and indexers:

```
private int counter;
public int Counter
{
    readonly get => counter;
    set => counter = value;
}
```

If you need to apply the `readonly` modifier to both accessors of a property or indexer, apply it in the declaration of the property or indexer.

NOTE

The compiler declares a `get` accessor of an [auto-implemented property](#) as `readonly`, regardless of presence of the `readonly` modifier in a property declaration.

In C# 9.0 and later, you may apply the `readonly` modifier to a property or indexer with an `init`

accessor:

```
public readonly double X { get; init; }
```

You can't apply the `readonly` modifier to static members of a structure type.

The compiler may make use of the `readonly` modifier for performance optimizations. For more information, see [Write safe and efficient C# code](#).

Nondestructive mutation

Beginning with C# 10, you can use the `with` expression to produce a copy of a structure-type instance with the specified properties and fields modified. You use [object initializer](#) syntax to specify what members to modify and their new values, as the following example shows:

```
public readonly struct Coords
{
    public Coords(double x, double y)
    {
        X = x;
        Y = y;
    }

    public double X { get; init; }
    public double Y { get; init; }

    public override string ToString() => $"({X}, {Y})";
}

public static void Main()
{
    var p1 = new Coords(0, 0);
    Console.WriteLine(p1); // output: (0, 0)

    var p2 = p1 with { X = 3 };
    Console.WriteLine(p2); // output: (3, 0)

    var p3 = p1 with { X = 1, Y = 4 };
    Console.WriteLine(p3); // output: (1, 4)
}
```

Limitations with the design of a structure type

When you design a structure type, you have the same capabilities as with a [class](#) type, with the following exceptions:

- You can't declare a parameterless constructor. Every structure type already provides an implicit parameterless constructor that produces the [default value](#) of the type.

NOTE

Beginning with C# 10, you can declare a parameterless constructor in a structure type. For more information, see the [Parameterless constructors and field initializers](#) section.

- You can't initialize an instance field or property at its declaration. However, you can initialize a [static](#) or [const](#) field or a static property at its declaration.

NOTE

Beginning with C# 10, you can initialize an instance field or property at its declaration. For more information, see the [Parameterless constructors and field initializers](#) section.

- A constructor of a structure type must initialize all instance fields of the type.
- A structure type can't inherit from other class or structure type and it can't be the base of a class. However, a structure type can implement [interfaces](#).
- You can't declare a [finalizer](#) within a structure type.

Parameterless constructors and field initializers

Beginning with C# 10, you can declare a parameterless instance constructor in a structure type, as the following example shows:

```
public readonly struct Measurement
{
    public Measurement()
    {
        Value = double.NaN;
        Description = "Undefined";
    }

    public Measurement(double value, string description)
    {
        Value = value;
        Description = description;
    }

    public double Value { get; init; }
    public string Description { get; init; }

    public override string ToString() => $"{Value} ({Description})";
}

public static void Main()
{
    var m1 = new Measurement();
    Console.WriteLine(m1); // output: NaN (Undefined)

    var m2 = default(Measurement);
    Console.WriteLine(m2); // output: 0 ()

    var ms = new Measurement[2];
    Console.WriteLine(string.Join(", ", ms)); // output: 0 (), 0 ()
}
```

As the preceding example shows, the [default value expression](#) ignores a parameterless constructor and produces the default value of a structure type, which is the value produced by setting all value-type fields to their [default values](#) (the 0-bit pattern) and all reference-type fields to `null`. Structure-type array instantiation also ignores a parameterless constructor and produces an array populated with the default values of a structure type.

Beginning with C# 10, you can also initialize an instance field or property at its declaration, as the following example shows:

```

public readonly struct Measurement
{
    public Measurement(double value)
    {
        Value = value;
    }

    public Measurement(double value, string description)
    {
        Value = value;
        Description = description;
    }

    public double Value { get; init; }
    public string Description { get; init; } = "Ordinary measurement";

    public override string ToString() => $"{Value} ({Description})";
}

public static void Main()
{
    var m1 = new Measurement(5);
    Console.WriteLine(m1); // output: 5 (Ordinary measurement)

    var m2 = new Measurement();
    Console.WriteLine(m2); // output: 0 ()

    var m3 = default(Measurement);
    Console.WriteLine(m3); // output: 0 ()
}

```

If you don't declare a parameterless constructor explicitly, a structure type provides a parameterless constructor whose behavior is as follows:

- If a structure type has explicit instance constructors or has no field initializers, an implicit parameterless constructor produces the default value of a structure type, regardless of field initializers, as the preceding example shows.
- If a structure type has no explicit instance constructors and has field initializers, the compiler synthesizes a public parameterless constructor that performs the specified field initializations, as the following example shows:

```

public struct Coords
{
    public double X = double.NaN;
    public double Y = double.NaN;

    public override string ToString() => $"({X}, {Y})";
}

public static void Main()
{
    var p1 = new Coords();
    Console.WriteLine(p1); // output: (NaN, NaN)

    var p2 = default(Coords);
    Console.WriteLine(p2); // output: (0, 0)

    var ps = new Coords[3];
    Console.WriteLine(string.Join(", ", ps)); // output: (0, 0), (0, 0), (0, 0)
}

```

As the preceding example shows, the default value expression and array instantiation ignore field initializers.

IMPORTANT

When a `struct` includes field initializers, but doesn't include any explicit instance constructors, the synthesized public parameterless constructor performs the specified field initializers. If that `struct` includes an explicit instance constructor, the synthesized parameterless constructor produces the same value as the `default` expression.

For more information, see the [Parameterless struct constructors](#) feature proposal note.

Instantiation of a structure type

In C#, you must initialize a declared variable before it can be used. Because a structure-type variable can't be `null` (unless it's a variable of a [nullable value type](#)), you must instantiate an instance of the corresponding type. There are several ways to do that.

Typically, you instantiate a structure type by calling an appropriate constructor with the `new` operator. Every structure type has at least one constructor. That's an implicit parameterless constructor, which produces the [default value](#) of the type. You can also use a [default value expression](#) to produce the default value of a type.

If all instance fields of a structure type are accessible, you can also instantiate it without the `new` operator. In that case you must initialize all instance fields before the first use of the instance. The following example shows how to do that:

```
public static class StructWithoutNew
{
    public struct Coords
    {
        public double x;
        public double y;
    }

    public static void Main()
    {
        Coords p;
        p.x = 3;
        p.y = 4;
        Console.WriteLine($"{p.x}, {p.y}"); // output: (3, 4)
    }
}
```

In the case of the [built-in value types](#), use the corresponding literals to specify a value of the type.

Passing structure-type variables by reference

When you pass a structure-type variable to a method as an argument or return a structure-type value from a method, the whole instance of a structure type is copied. That can affect the performance of your code in high-performance scenarios that involve large structure types. You can avoid value copying by passing a structure-type variable by reference. Use the `ref`, `out`, or `in` method parameter modifiers to indicate that an argument must be passed by reference. Use [ref returns](#) to return a method result by reference. For more information, see [Write safe and efficient C# code](#).

`ref` struct

Beginning with C# 7.2, you can use the `ref` modifier in the declaration of a structure type. Instances of a `ref` struct type are allocated on the stack and can't escape to the managed heap. To ensure that, the compiler limits the usage of `ref` struct types as follows:

- A `ref` struct can't be the element type of an array.
- A `ref` struct can't be a declared type of a field of a class or a non-`ref` struct.
- A `ref` struct can't implement interfaces.
- A `ref` struct can't be boxed to `System.ValueType` or `System.Object`.
- A `ref` struct can't be a type argument.
- A `ref` struct variable can't be captured by a [lambda expression](#) or a [local function](#).
- A `ref` struct variable can't be used in an `async` method. However, you can use `ref` struct variables in synchronous methods, for example, in those that return `Task` or `Task<TResult>`.
- A `ref` struct variable can't be used in [iterators](#).

Beginning with C# 8.0, you can define a disposable `ref` struct. To do that, ensure that a `ref` struct fits the [disposable pattern](#). That is, it has an instance or extension `Dispose` method, which is accessible, parameterless and has a `void` return type.

Typically, you define a `ref` struct type when you need a type that also includes data members of `ref` struct types:

```
public ref struct CustomRef
{
    public bool IsValid;
    public Span<int> Inputs;
    public Span<int> Outputs;
}
```

To declare a `ref` struct as [readonly](#), combine the `readonly` and `ref` modifiers in the type declaration (the `readonly` modifier must come before the `ref` modifier):

```
public readonly ref struct ConversionRequest
{
    public ConversionRequest(double rate, ReadOnlySpan<double> values)
    {
        Rate = rate;
        Values = values;
    }

    public double Rate { get; }
    public ReadOnlySpan<double> Values { get; }
}
```

In .NET, examples of a `ref` struct are [System.Span<T>](#) and [System.ReadOnlySpan<T>](#).

struct constraint

You also use the `struct` keyword in the [struct constraint](#) to specify that a type parameter is a non-nullable value type. Both structure and [enumeration](#) types satisfy the `struct` constraint.

Conversions

For any structure type (except `ref struct` types), there exist [boxing and unboxing](#) conversions to and from the `System.ValueType` and `System.Object` types. There exist also boxing and unboxing conversions between a structure type and any interface that it implements.

C# language specification

For more information, see the [Structs](#) section of the [C# language specification](#).

For more information about features introduced in C# 7.2 and later, see the following feature proposal notes:

- [Readonly structs](#)
- [Readonly instance members](#)
- [Compile-time safety for ref-like types](#)
- [Parameterless struct constructors](#)
- [Allow `with` expression on structs](#)

See also

- [C# reference](#)
- [Design guidelines - Choosing between class and struct](#)
- [Design guidelines - Struct design](#)
- [The C# type system](#)

Tuple types (C# reference)

12/28/2021 • 8 minutes to read • [Edit Online](#)

Available in C# 7.0 and later, the *tuples* feature provides concise syntax to group multiple data elements in a lightweight data structure. The following example shows how you can declare a tuple variable, initialize it, and access its data members:

```
(double, int) t1 = (4.5, 3);
Console.WriteLine($"Tuple with elements {t1.Item1} and {t1.Item2}.");
// Output:
// Tuple with elements 4.5 and 3.

(double Sum, int Count) t2 = (4.5, 3);
Console.WriteLine($"Sum of {t2.Count} elements is {t2.Sum}.");
// Output:
// Sum of 3 elements is 4.5.
```

As the preceding example shows, to define a tuple type, you specify types of all its data members and, optionally, the [field names](#). You cannot define methods in a tuple type, but you can use the methods provided by .NET, as the following example shows:

```
(double, int) t = (4.5, 3);
Console.WriteLine(t.ToString());
Console.WriteLine($"Hash code of {t} is {t.GetHashCode()}.");
// Output:
// (4.5, 3)
// Hash code of (4.5, 3) is 718460086.
```

Beginning with C# 7.3, tuple types support [equality operators](#) `==` and `!=`. For more information, see the [Tuple equality](#) section.

Tuple types are [value types](#); tuple elements are public fields. That makes tuples *mutable* value types.

NOTE

The tuples feature requires the [System.ValueTuple](#) type and related generic types (for example, [System.ValueTuple<T1,T2>](#)), which are available in .NET Core and .NET Framework 4.7 and later. To use tuples in a project that targets .NET Framework 4.6.2 or earlier, add the NuGet package [System.ValueTuple](#) to the project.

You can define tuples with an arbitrary large number of elements:

```
var t =
(1, 2, 3, 4, 5, 6, 7, 8, 9, 10,
11, 12, 13, 14, 15, 16, 17, 18,
19, 20, 21, 22, 23, 24, 25, 26);
Console.WriteLine(t.Item26); // output: 26
```

Use cases of tuples

One of the most common use cases of tuples is as a method return type. That is, instead of defining [out](#) [method parameters](#), you can group method results in a tuple return type, as the following example shows:

```

var xs = new[] { 4, 7, 9 };
var limits = FindMinMax(xs);
Console.WriteLine($"Limits of [{string.Join(" ", xs)}] are {limits.min} and {limits.max}");
// Output:
// Limits of [4 7 9] are 4 and 9

var ys = new[] { -9, 0, 67, 100 };
var (minimum, maximum) = FindMinMax(ys);
Console.WriteLine($"Limits of [{string.Join(" ", ys)}] are {minimum} and {maximum}");
// Output:
// Limits of [-9 0 67 100] are -9 and 100

(int min, int max) FindMinMax(int[] input)
{
    if (input is null || input.Length == 0)
    {
        throw new ArgumentException("Cannot find minimum and maximum of a null or empty array.");
    }

    var min = int.MaxValue;
    var max = int.MinValue;
    foreach (var i in input)
    {
        if (i < min)
        {
            min = i;
        }
        if (i > max)
        {
            max = i;
        }
    }
    return (min, max);
}

```

As the preceding example shows, you can work with the returned tuple instance directly or [deconstruct](#) it in separate variables.

You can also use tuple types instead of [anonymous types](#); for example, in LINQ queries. For more information, see [Choosing between anonymous and tuple types](#).

Typically, you use tuples to group loosely related data elements. That is usually useful within private and internal utility methods. In the case of public API, consider defining a [class](#) or a [structure](#) type.

Tuple field names

You can explicitly specify the names of tuple fields either in a tuple initialization expression or in the definition of a tuple type, as the following example shows:

```

var t = (Sum: 4.5, Count: 3);
Console.WriteLine($"Sum of {t.Count} elements is {t.Sum}.");

(double Sum, int Count) d = (4.5, 3);
Console.WriteLine($"Sum of {d.Count} elements is {d.Sum}.");

```

Beginning with C# 7.1, if you don't specify a field name, it may be inferred from the name of the corresponding variable in a tuple initialization expression, as the following example shows:

```
var sum = 4.5;
var count = 3;
var t = (sum, count);
Console.WriteLine($"Sum of {t.count} elements is {t.sum}.");
```

That's known as tuple projection initializers. The name of a variable isn't projected onto a tuple field name in the following cases:

- The candidate name is a member name of a tuple type, for example, `Item3`, `ToString`, or `Rest`.
- The candidate name is a duplicate of another tuple field name, either explicit or implicit.

In those cases you either explicitly specify the name of a field or access a field by its default name.

The default names of tuple fields are `Item1`, `Item2`, `Item3` and so on. You can always use the default name of a field, even when a field name is specified explicitly or inferred, as the following example shows:

```
var a = 1;
var t = (a, b: 2, 3);
Console.WriteLine($"The 1st element is {t.Item1} (same as {t.a}).");
Console.WriteLine($"The 2nd element is {t.Item2} (same as {t.b}).");
Console.WriteLine($"The 3rd element is {t.Item3}.");
// Output:
// The 1st element is 1 (same as 1).
// The 2nd element is 2 (same as 2).
// The 3rd element is 3.
```

[Tuple assignment](#) and [tuple equality comparisons](#) don't take field names into account.

At compile time, the compiler replaces non-default field names with the corresponding default names. As a result, explicitly specified or inferred field names aren't available at run time.

Tuple assignment and deconstruction

C# supports assignment between tuple types that satisfy both of the following conditions:

- both tuple types have the same number of elements
- for each tuple position, the type of the right-hand tuple element is the same as or implicitly convertible to the type of the corresponding left-hand tuple element

Tuple element values are assigned following the order of tuple elements. The names of tuple fields are ignored and not assigned, as the following example shows:

```
(int, double) t1 = (17, 3.14);
(double First, double Second) t2 = (0.0, 1.0);
t2 = t1;
Console.WriteLine($"{nameof(t2)}: {t2.First} and {t2.Second}");
// Output:
// t2: 17 and 3.14

(double A, double B) t3 = (2.0, 3.0);
t3 = t2;
Console.WriteLine($"{nameof(t3)}: {t3.A} and {t3.B}");
// Output:
// t3: 17 and 3.14
```

You can also use the assignment operator `=` to *deconstruct* a tuple instance in separate variables. You can do that in one of the following ways:

- Explicitly declare the type of each variable inside parentheses:

```
var t = ("post office", 3.6);
(string destination, double distance) = t;
Console.WriteLine($"Distance to {destination} is {distance} kilometers.");
// Output:
// Distance to post office is 3.6 kilometers.
```

- Use the `var` keyword outside the parentheses to declare implicitly typed variables and let the compiler infer their types:

```
var t = ("post office", 3.6);
var (destination, distance) = t;
Console.WriteLine($"Distance to {destination} is {distance} kilometers.");
// Output:
// Distance to post office is 3.6 kilometers.
```

- Use existing variables:

```
var destination = string.Empty;
var distance = 0.0;

var t = ("post office", 3.6);
(destination, distance) = t;
Console.WriteLine($"Distance to {destination} is {distance} kilometers.");
// Output:
// Distance to post office is 3.6 kilometers.
```

For more information about deconstruction of tuples and other types, see [Deconstructing tuples and other types](#).

Tuple equality

Beginning with C# 7.3, tuple types support the `==` and `!=` operators. These operators compare members of the left-hand operand with the corresponding members of the right-hand operand following the order of tuple elements.

```
(int a, byte b) left = (5, 10);
(long a, int b) right = (5, 10);
Console.WriteLine(left == right); // output: True
Console.WriteLine(left != right); // output: False

var t1 = (A: 5, B: 10);
var t2 = (B: 5, A: 10);
Console.WriteLine(t1 == t2); // output: True
Console.WriteLine(t1 != t2); // output: False
```

As the preceding example shows, the `==` and `!=` operations don't take into account tuple field names.

Two tuples are comparable when both of the following conditions are satisfied:

- Both tuples have the same number of elements. For example, `t1 != t2` doesn't compile if `t1` and `t2` have different numbers of elements.
- For each tuple position, the corresponding elements from the left-hand and right-hand tuple operands are comparable with the `==` and `!=` operators. For example, `(1, (2, 3)) == ((1, 2), 3)` doesn't compile because `1` is not comparable with `(1, 2)`.

The `==` and `!=` operators compare tuples in short-circuiting way. That is, an operation stops as soon as it meets a pair of non equal elements or reaches the ends of tuples. However, before any comparison, *all* tuple elements are evaluated, as the following example shows:

```
Console.WriteLine((Display(1), Display(2)) == (Display(3), Display(4)));

int Display(int s)
{
    Console.WriteLine(s);
    return s;
}

// Output:
// 1
// 2
// 3
// 4
// False
```

Tuples as out parameters

Typically, you refactor a method that has `out` parameters into a method that returns a tuple. However, there are cases in which an `out` parameter can be of a tuple type. The following example shows how to work with tuples as `out` parameters:

```
var limitsLookup = new Dictionary<int, (int Min, int Max)>()
{
    [2] = (4, 10),
    [4] = (10, 20),
    [6] = (0, 23)
};

if (limitsLookup.TryGetValue(4, out (int Min, int Max) limits))
{
    Console.WriteLine($"Found limits: min is {limits.Min}, max is {limits.Max}");
}

// Output:
// Found limits: min is 10, max is 20
```

Tuples vs `System.Tuple`

C# tuples, which are backed by `System.ValueTuple` types, are different from tuples that are represented by `System.Tuple` types. The main differences are as follows:

- `System.ValueTuple` types are [value types](#). `System.Tuple` types are [reference types](#).
- `System.ValueTuple` types are mutable. `System.Tuple` types are immutable.
- Data members of `System.ValueTuple` types are fields. Data members of `System.Tuple` types are properties.

C# language specification

For more information, see the following feature proposal notes:

- [Infer tuple names \(aka. tuple projection initializers\)](#)
- [Support for `==` and `!=` on tuple types](#)

See also

- [C# reference](#)

- Value types
- Choosing between anonymous and tuple types
- `System.ValueTuple`

Nullable value types (C# reference)

12/28/2021 • 7 minutes to read • [Edit Online](#)

A *nullable value type* `T?` represents all values of its underlying *value type* `T` and an additional *null* value. For example, you can assign any of the following three values to a `bool?` variable: `true`, `false`, or `null`. An underlying value type `T` cannot be a nullable value type itself.

NOTE

C# 8.0 introduces the nullable reference types feature. For more information, see [Nullable reference types](#). The nullable value types are available beginning with C# 2.

Any nullable value type is an instance of the generic `System.Nullable<T>` structure. You can refer to a nullable value type with an underlying type `T` in any of the following interchangeable forms: `Nullable<T>` or `T?`.

You typically use a nullable value type when you need to represent the undefined value of an underlying value type. For example, a Boolean, or `bool`, variable can only be either `true` or `false`. However, in some applications a variable value can be undefined or missing. For example, a database field may contain `true` or `false`, or it may contain no value at all, that is, `NULL`. You can use the `bool?` type in that scenario.

Declaration and assignment

As a value type is implicitly convertible to the corresponding nullable value type, you can assign a value to a variable of a nullable value type as you would do that for its underlying value type. You can also assign the `null` value. For example:

```
double? pi = 3.14;
char? letter = 'a';

int m2 = 10;
int? m = m2;

bool? flag = null;

// An array of a nullable value type:
int?[] arr = new int?[10];
```

The default value of a nullable value type represents `null`, that is, it's an instance whose `Nullable<T>.HasValue` property returns `false`.

Examination of an instance of a nullable value type

Beginning with C# 7.0, you can use the `is` [operator with a type pattern](#) to both examine an instance of a nullable value type for `null` and retrieve a value of an underlying type:

```
int? a = 42;
if (a is int valueOfA)
{
    Console.WriteLine($"a is {valueOfA}");
}
else
{
    Console.WriteLine("a does not have a value");
}
// Output:
// a is 42
```

You always can use the following read-only properties to examine and get a value of a nullable value type variable:

- [Nullable<T>.HasValue](#) indicates whether an instance of a nullable value type has a value of its underlying type.
- [Nullable<T>.Value](#) gets the value of an underlying type if [HasValue](#) is `true`. If [HasValue](#) is `false`, the [Value](#) property throws an [InvalidOperationException](#).

The following example uses the `HasValue` property to test whether the variable contains a value before displaying it:

```
int? b = 10;
if (b.HasValue)
{
    Console.WriteLine($"b is {b.Value}");
}
else
{
    Console.WriteLine("b does not have a value");
}
// Output:
// b is 10
```

You can also compare a variable of a nullable value type with `null` instead of using the `HasValue` property, as the following example shows:

```
int? c = 7;
if (c != null)
{
    Console.WriteLine($"c is {c.Value}");
}
else
{
    Console.WriteLine("c does not have a value");
}
// Output:
// c is 7
```

Conversion from a nullable value type to an underlying type

If you want to assign a value of a nullable value type to a non-nullable value type variable, you might need to specify the value to be assigned in place of `null`. Use the [null-coalescing operator](#) `??` to do that (you can also use the [Nullable<T>.GetValueOrDefault\(T\)](#) method for the same purpose):

```
int? a = 28;
int b = a ?? -1;
Console.WriteLine($"b is {b}"); // output: b is 28

int? c = null;
int d = c ?? -1;
Console.WriteLine($"d is {d}"); // output: d is -1
```

If you want to use the [default](#) value of the underlying value type in place of `null`, use the [Nullable<T>.GetValueOrDefault\(\)](#) method.

You can also explicitly cast a nullable value type to a non-nullable type, as the following example shows:

```
int? n = null;

//int m1 = n;    // Doesn't compile
int n2 = (int)n; // Compiles, but throws an exception if n is null
```

At run time, if the value of a nullable value type is `null`, the explicit cast throws an [InvalidOperationException](#).

A non-nullable value type `T` is implicitly convertible to the corresponding nullable value type `T?`.

Lifted operators

The predefined unary and binary [operators](#) or any overloaded operators that are supported by a value type `T` are also supported by the corresponding nullable value type `T?`. These operators, also known as *lifted operators*, produce `null` if one or both operands are `null`; otherwise, the operator uses the contained values of its operands to calculate the result. For example:

```
int? a = 10;
int? b = null;
int? c = 10;

a++;           // a is 11
a = a * c;     // a is 110
a = a + b;     // a is null
```

NOTE

For the `bool?` type, the predefined `&` and `|` operators don't follow the rules described in this section: the result of an operator evaluation can be non-null even if one of the operands is `null`. For more information, see the [Nullable Boolean logical operators](#) section of the [Boolean logical operators](#) article.

For the [comparison operators](#) `<`, `>`, `<=`, and `>=`, if one or both operands are `null`, the result is `false`; otherwise, the contained values of operands are compared. Do not assume that because a particular comparison (for example, `<=`) returns `false`, the opposite comparison (`>`) returns `true`. The following example shows that 10 is

- neither greater than or equal to `null`
- nor less than `null`

```

int? a = 10;
Console.WriteLine($"{a} >= null is {a >= null}");
Console.WriteLine($"{a} < null is {a < null}");
Console.WriteLine($"{a} == null is {a == null}");
// Output:
// 10 >= null is False
// 10 < null is False
// 10 == null is False

int? b = null;
int? c = null;
Console.WriteLine($"null >= null is {b >= c}");
Console.WriteLine($"null == null is {b == c}");
// Output:
// null >= null is False
// null == null is True

```

For the [equality operator](#) `==`, if both operands are `null`, the result is `true`, if only one of the operands is `null`, the result is `false`; otherwise, the contained values of operands are compared.

For the [inequality operator](#) `!=`, if both operands are `null`, the result is `false`, if only one of the operands is `null`, the result is `true`; otherwise, the contained values of operands are compared.

If there exists a [user-defined conversion](#) between two value types, the same conversion can also be used between the corresponding nullable value types.

Boxing and unboxing

An instance of a nullable value type `T?` is [boxed](#) as follows:

- If [HasValue](#) returns `false`, the null reference is produced.
- If [HasValue](#) returns `true`, the corresponding value of the underlying value type `T` is boxed, not the instance of [Nullable<T>](#).

You can unbox a boxed value of a value type `T` to the corresponding nullable value type `T?`, as the following example shows:

```

int a = 41;
object aBoxed = a;
int? aNullable = (int?)aBoxed;
Console.WriteLine($"Value of aNullable: {aNullable}");

object aNullableBoxed = aNullable;
if (aNullableBoxed is int valueOfA)
{
    Console.WriteLine($"aNullableBoxed is boxed int: {valueOfA}");
}
// Output:
// Value of aNullable: 41
// aNullableBoxed is boxed int: 41

```

How to identify a nullable value type

The following example shows how to determine whether a [System.Type](#) instance represents a constructed nullable value type, that is, the [System.Nullable<T>](#) type with a specified type parameter `T`:

```

Console.WriteLine($"int? is {(Nullable.GetTypeOf(int?)) ? "nullable" : "non nullable"} value type");
Console.WriteLine($"int is {(Nullable.GetTypeOf(int)) ? "nullable" : "non-nullable"} value type");

bool IsNullable(Type type) => Nullable.GetUnderlyingType(type) != null;

// Output:
// int? is nullable value type
// int is non-nullable value type

```

As the example shows, you use the `typeof` operator to create a `System.Type` instance.

If you want to determine whether an instance is of a nullable value type, don't use the `Object.GetType` method to get a `Type` instance to be tested with the preceding code. When you call the `Object.GetType` method on an instance of a nullable value type, the instance is boxed to `Object`. As boxing of a non-null instance of a nullable value type is equivalent to boxing of a value of the underlying type, `GetType` returns a `Type` instance that represents the underlying type of a nullable value type:

```

int? a = 17;
Type typeOfA = a.GetType();
Console.WriteLine(typeOfA.FullName);
// Output:
// System.Int32

```

Also, don't use the `is` operator to determine whether an instance is of a nullable value type. As the following example shows, you cannot distinguish types of a nullable value type instance and its underlying type instance with the `is` operator:

```

int? a = 14;
if (a is int)
{
    Console.WriteLine("int? instance is compatible with int");
}

int b = 17;
if (b is int?)
{
    Console.WriteLine("int instance is compatible with int?");
}
// Output:
// int? instance is compatible with int
// int instance is compatible with int?

```

You can use the code presented in the following example to determine whether an instance is of a nullable value type:

```

int? a = 14;
Console.WriteLine(IsOfNullableType(a)); // output: True

int b = 17;
Console.WriteLine(IsOfNullableType(b)); // output: False

bool IsOfNullableType<T>(T o)
{
    var type = typeof(T);
    return Nullable.GetUnderlyingType(type) != null;
}

```

NOTE

The methods described in this section are not applicable in the case of [nullable reference types](#).

C# language specification

For more information, see the following sections of the [C# language specification](#):

- [Nullable types](#)
- [Lifted operators](#)
- [Implicit nullable conversions](#)
- [Explicit nullable conversions](#)
- [Lifted conversion operators](#)

See also

- [C# reference](#)
- [What exactly does 'lifted' mean?](#)
- [System.Nullable<T>](#)
- [System.Nullable](#)
- [Nullable.GetUnderlyingType](#)
- [Nullable reference types](#)

Reference types (C# Reference)

12/28/2021 • 2 minutes to read • [Edit Online](#)

There are two kinds of types in C#: reference types and value types. Variables of reference types store references to their data (objects), while variables of value types directly contain their data. With reference types, two variables can reference the same object; therefore, operations on one variable can affect the object referenced by the other variable. With value types, each variable has its own copy of the data, and it is not possible for operations on one variable to affect the other (except in the case of `in`, `ref` and `out` parameter variables; see [in](#), [ref](#) and [out](#) parameter modifier).

The following keywords are used to declare reference types:

- [class](#)
- [interface](#)
- [delegate](#)
- [record](#)

C# also provides the following built-in reference types:

- [dynamic](#)
- [object](#)
- [string](#)

See also

- [C# Reference](#)
- [C# Keywords](#)
- [Pointer types](#)
- [Value types](#)

Built-in reference types (C# reference)

12/28/2021 • 7 minutes to read • [Edit Online](#)

C# has a number of built-in reference types. They have keywords or operators that are synonyms for a type in the .NET library.

The object type

The `object` type is an alias for `System.Object` in .NET. In the unified type system of C#, all types, predefined and user-defined, reference types and value types, inherit directly or indirectly from `System.Object`. You can assign values of any type to variables of type `object`. Any `object` variable can be assigned to its default value using the literal `null`. When a variable of a value type is converted to object, it is said to be *boxed*. When a variable of type `object` is converted to a value type, it is said to be *unboxed*. For more information, see [Boxing and Unboxing](#).

The string type

The `string` type represents a sequence of zero or more Unicode characters. `string` is an alias for `System.String` in .NET.

Although `string` is a reference type, the [equality operators](#) `==` and `!=` are defined to compare the values of `string` objects, not references. This makes testing for string equality more intuitive. For example:

```
string a = "hello";
string b = "h";
// Append to contents of 'b'
b += "ello";
Console.WriteLine(a == b);
Console.WriteLine(object.ReferenceEquals(a, b));
```

This displays "True" and then "False" because the content of the strings are equivalent, but `a` and `b` do not refer to the same string instance.

The [+ operator](#) concatenates strings:

```
string a = "good " + "morning";
```

This creates a string object that contains "good morning".

Strings are *immutable*--the contents of a string object cannot be changed after the object is created, although the syntax makes it appear as if you can do this. For example, when you write this code, the compiler actually creates a new string object to hold the new sequence of characters, and that new object is assigned to `b`. The memory that had been allocated for `b` (when it contained the string "h") is then eligible for garbage collection.

```
string b = "h";
b += "ello";
```

The `[]` [operator](#) can be used for readonly access to individual characters of a string. Valid index values start at `0` and must be less than the length of the string:


```
string str = "test";
char x = str[2]; // x = 's';
```

In similar fashion, the `[]` operator can also be used for iterating over each character in a string:

```
string str = "test";

for (int i = 0; i < str.Length; i++)
{
    Console.Write(str[i] + " ");
}
// Output: t e s t
```

String literals are of type `string` and can be written in two forms, quoted and `@`-quoted. Quoted string literals are enclosed in double quotation marks (`"`):

```
"good morning" // a string literal
```

String literals can contain any character literal. Escape sequences are included. The following example uses escape sequence `\\` for backslash, `\u0066` for the letter `f`, and `\n` for newline.

```
string a = "\\u0066\n F";
Console.WriteLine(a);
// Output:
// \f
// F
```

NOTE

The escape code `\dddd` (where `dddd` is a four-digit number) represents the Unicode character U+`dddd`. Eight-digit Unicode escape codes are also recognized: `\Udddddddd`.

Verbatim string literals start with `@` and are also enclosed in double quotation marks. For example:

```
@"good morning" // a string literal
```

The advantage of verbatim strings is that escape sequences are *not* processed, which makes it easy to write, for example, a fully qualified Windows file name:

```
@"c:\Docs\Source\a.txt" // rather than "c:\\Docs\\Source\\a.txt"
```

To include a double quotation mark in an `@`-quoted string, double it:

```
@"""Ahoy!""" cried the captain." // "Ahoy!" cried the captain.
```

The delegate type

The declaration of a delegate type is similar to a method signature. It has a return value and any number of parameters of any type:

```
public delegate void MessageDelegate(string message);
public delegate int AnotherDelegate(MyType m, long num);
```

In .NET, `System.Action` and `System.Func` types provide generic definitions for many common delegates. You likely don't need to define new custom delegate types. Instead, you can create instantiations of the provided generic types.

A `delegate` is a reference type that can be used to encapsulate a named or an anonymous method. Delegates are similar to function pointers in C++; however, delegates are type-safe and secure. For applications of delegates, see [Delegates](#) and [Generic Delegates](#). Delegates are the basis for [Events](#). A delegate can be instantiated by associating it either with a named or anonymous method.

The delegate must be instantiated with a method or lambda expression that has a compatible return type and input parameters. For more information on the degree of variance that is allowed in the method signature, see [Variance in Delegates](#). For use with anonymous methods, the delegate and the code to be associated with it are declared together.

Delegate combination and removal fails with a runtime exception when the delegate types involved at run time are different due to variant conversion. The following example demonstrates a situation which fails:

```
Action<string> stringAction = str => {};
Action<object> objectAction = obj => {};

// Valid due to implicit reference conversion of
// objectAction to Action<string>, but may fail
// at run time.
Action<string> combination = stringAction + objectAction;
```

You can create a delegate with the correct runtime type by creating a new delegate object. The following example demonstrates how this workaround may be applied to the preceding example.

```
Action<string> stringAction = str => {};
Action<object> objectAction = obj => {};

// Creates a new delegate instance with a runtime type of Action<string>.
Action<string> wrappedObjectAction = new Action<string>(objectAction);

// The two Action<string> delegate instances can now be combined.
Action<string> combination = stringAction + wrappedObjectAction;
```

Beginning with C# 9, you can declare [function pointers](#), which use similar syntax. A function pointer uses the `calli` instruction instead of instantiating a delegate type and calling the virtual `Invoke` method.

The dynamic type

The `dynamic` type indicates that use of the variable and references to its members bypass compile-time type checking. Instead, these operations are resolved at run time. The `dynamic` type simplifies access to COM APIs such as the Office Automation APIs, to dynamic APIs such as IronPython libraries, and to the HTML Document Object Model (DOM).

Type `dynamic` behaves like type `object` in most circumstances. In particular, any non-null expression can be converted to the `dynamic` type. The `dynamic` type differs from `object` in that operations that contain expressions of type `dynamic` are not resolved or type checked by the compiler. The compiler packages together information about the operation, and that information is later used to evaluate the operation at run time. As part of the process, variables of type `dynamic` are compiled into variables of type `object`. Therefore, type `dynamic`

exists only at compile time, not at run time.

The following example contrasts a variable of type `dynamic` to a variable of type `object`. To verify the type of each variable at compile time, place the mouse pointer over `dyn` or `obj` in the `WriteLine` statements. Copy the following code into an editor where IntelliSense is available. IntelliSense shows **dynamic** for `dyn` and **object** for `obj`.

```
class Program
{
    static void Main(string[] args)
    {
        dynamic dyn = 1;
        object obj = 1;

        // Rest the mouse pointer over dyn and obj to see their
        // types at compile time.
        System.Console.WriteLine(dyn.GetType());
        System.Console.WriteLine(obj.GetType());
    }
}
```

The `WriteLine` statements display the run-time types of `dyn` and `obj`. At that point, both have the same type, integer. The following output is produced:

```
System.Int32
System.Int32
```

To see the difference between `dyn` and `obj` at compile time, add the following two lines between the declarations and the `WriteLine` statements in the previous example.

```
dyn = dyn + 3;
obj = obj + 3;
```

A compiler error is reported for the attempted addition of an integer and an object in expression `obj + 3`. However, no error is reported for `dyn + 3`. The expression that contains `dyn` is not checked at compile time because the type of `dyn` is `dynamic`.

The following example uses `dynamic` in several declarations. The `Main` method also contrasts compile-time type checking with run-time type checking.

```

using System;

namespace DynamicExamples
{
    class Program
    {
        static void Main(string[] args)
        {
            ExampleClass ec = new ExampleClass();
            Console.WriteLine(ec.exampleMethod(10));
            Console.WriteLine(ec.exampleMethod("value"));

            // The following line causes a compiler error because exampleMethod
            // takes only one argument.
            //Console.WriteLine(ec.exampleMethod(10, 4));

            dynamic dynamic_ec = new ExampleClass();
            Console.WriteLine(dynamic_ec.exampleMethod(10));

            // Because dynamic_ec is dynamic, the following call to exampleMethod
            // with two arguments does not produce an error at compile time.
            // However, it does cause a run-time error.
            //Console.WriteLine(dynamic_ec.exampleMethod(10, 4));
        }
    }

    class ExampleClass
    {
        static dynamic field;
        dynamic prop { get; set; }

        public dynamic exampleMethod(dynamic d)
        {
            dynamic local = "Local variable";
            int two = 2;

            if (d is int)
            {
                return local;
            }
            else
            {
                return two;
            }
        }
    }
}

// Results:
// Local variable
// 2
// Local variable

```

See also

- [C# Reference](#)
- [C# Keywords](#)
- [Events](#)
- [Using Type dynamic](#)
- [Best Practices for Using Strings](#)
- [Basic String Operations](#)
- [Creating New Strings](#)
- [Type-testing and cast operators](#)
- [How to safely cast using pattern matching and the as and is operators](#)

- [Walkthrough: creating and using dynamic objects](#)
- [System.Object](#)
- [System.String](#)
- [System.Dynamic.DynamicObject](#)

Records (C# reference)

12/28/2021 • 17 minutes to read • [Edit Online](#)

Beginning with C# 9, you use the `record` keyword to define a [reference type](#) that provides built-in functionality for encapsulating data. You can create record types with immutable properties by using positional parameters or standard property syntax:

```
public record Person(string FirstName, string LastName);
```

```
public record Person
{
    public string FirstName { get; init; } = default!;
    public string LastName { get; init; } = default!;
};
```

You can also create record types with mutable properties and fields:

```
public record Person
{
    public string FirstName { get; set; } = default!;
    public string LastName { get; set; } = default!;
};
```

While records can be mutable, they're primarily intended for supporting immutable data models. The record type offers the following features:

- [Concise syntax for creating a reference type with immutable properties](#)
- Built-in behavior useful for a data-centric reference type:
 - [Value equality](#)
 - [Concise syntax for nondestructive mutation](#)
 - [Built-in formatting for display](#)
- [Support for inheritance hierarchies](#)

You can also use [structure types](#) to design data-centric types that provide value equality and little or no behavior. In C# 10 and later, you can define `record struct` types using either positional parameters, or standard property syntax:

```
public readonly record struct Point(double X, double Y, double Z);
```

```
public record struct Point
{
    public double X { get; init; }
    public double Y { get; init; }
    public double Z { get; init; }
}
```

Record structs can be mutable as well, both positional record structs and record structs with no positional parameters:

```
public record struct DataMeasurement(DateTime TakenAt, double Measurement);
```

```
public record struct Point
{
    public double X { get; set; }
    public double Y { get; set; }
    public double Z { get; set; }
}
```

The preceding examples show some distinctions between records that are reference types and records that are value types:

- A `record` or a `record class` declares a reference type. The `class` keyword is optional, but can add clarity for readers. A `record struct` declares a value type.
- Positional properties are *immutable* in a `record class` and a `readonly record struct`. They're *mutable* in a `record struct`.

The remainder of this article discusses both `record class` and `record struct` types. The differences are detailed in each section. You should decide between a `record class` and a `record struct` similar to deciding between a `class` and a `struct`. The term *record* is used to describe behavior that applies to all record types. Either `record struct` or `record class` is used to describe behavior that applies to only struct or class types, respectively.

Positional syntax for property definition

You can use positional parameters to declare properties of a record and to initialize the property values when you create an instance:

```
public record Person(string FirstName, string LastName);

public static void Main()
{
    Person person = new("Nancy", "Davolio");
    Console.WriteLine(person);
    // output: Person { FirstName = Nancy, LastName = Davolio }
}
```

When you use the positional syntax for property definition, the compiler creates:

- A public auto-implemented property for each positional parameter provided in the record declaration.
 - For `record` types and `readonly record struct` types: An *init-only* property.
 - For `record struct` types: A read-write property.
- A primary constructor whose parameters match the positional parameters on the record declaration.
- For record struct types, a parameterless constructor that sets each field to its default value.
- A `Deconstruct` method with an `out` parameter for each positional parameter provided in the record declaration. The method deconstructs properties defined by using positional syntax; it ignores properties that are defined by using standard property syntax.

You may want to add attributes to any of these elements the compiler creates from the record definition. You can add a *target* to any attribute you apply to the positional record's properties. The following example applies the [System.Text.Json.Serialization.JsonPropertyNameAttribute](#) to each property of the `Person` record. The `property:` target indicates that the attribute is applied to the compiler-generated property. Other values are `field:` to apply the attribute to the field, and `param:` to apply the attribute to the parameter.

```

/// <summary>
/// Person record type
/// </summary>
/// <param name="FirstName">First Name</param>
/// <param name="LastName">Last Name</param>
/// <remarks>
/// The person type is a positional record containing the
/// properties for the first and last name. Those properties
/// map to the JSON elements "firstName" and "lastName" when
/// serialized or deserialized.
/// </remarks>
public record Person([property: JsonPropertyName("firstName")]string FirstName,
    [property: JsonPropertyName("lastName")]string LastName);

```

The preceding example also shows how to create XML documentation comments for the record. You can add the `<param>` tag to add documentation for the primary constructor's parameters.

If the generated auto-implemented property definition isn't what you want, you can define your own property of the same name. For example, you may want to change accessibility or mutability, or provide an implementation for either the `get` or `set` accessor. If you declare the property in your source, you must initialize it from the positional parameter of the record. The generated deconstructor will use your property definition. For instance, the following example declares the `FirstName` and `LastName` properties of a positional record `public`, but restricts the `Id` positional parameter to `internal`. You can use this syntax for records and record struct types.

```

public record Person(string FirstName, string LastName, string Id)
{
    internal string Id { get; init; } = Id;
}

public static void Main()
{
    Person person = new("Nancy", "Davolio", "12345");
    Console.WriteLine(person.FirstName); //output: Nancy
}

```

A record type doesn't have to declare any positional properties. You can declare a record without any positional properties, and you can declare other fields and properties, as in the following example:

```

public record Person(string FirstName, string LastName)
{
    public string[] PhoneNumbers { get; init; } = Array.Empty<string>();
};

```

If you define properties by using standard property syntax but omit the access modifier, the properties are implicitly `private`.

Immutability

A *positional record* and a *positional readonly record struct* declare init-only properties. A *positional record struct* declares read-write properties. You can override either of those defaults, as shown in the previous section.

Immutability can be useful when you need a data-centric type to be thread-safe or you're depending on a hash code remaining the same in a hash table. Immutability isn't appropriate for all data scenarios, however. [Entity Framework Core](#), for example, doesn't support updating with immutable entity types.

Init-only properties, whether created from positional parameters (`record class` , and `readonly record struct`) or by specifying `init` accessors, have *shallow immutability*. After initialization, you can't change the value of value-type properties or the reference of reference-type properties. However, the data that a reference-type property refers to can be changed. The following example shows that the content of a reference-type immutable property (an array in this case) is mutable:

```
public record Person(string FirstName, string LastName, string[] PhoneNumbers);

public static void Main()
{
    Person person = new("Nancy", "Davolio", new string[1] { "555-1234" });
    Console.WriteLine(person.PhoneNumbers[0]); // output: 555-1234

    person.PhoneNumbers[0] = "555-6789";
    Console.WriteLine(person.PhoneNumbers[0]); // output: 555-6789
}
```

The features unique to record types are implemented by compiler-synthesized methods, and none of these methods compromises immutability by modifying object state. Unless specified, the synthesized methods are generated for `record` , `record struct` , and `readonly record struct` declarations.

Value equality

For any type you define, you can override `Object.Equals(Object)`, and overload `operator ==` . If you don't override `Equals` or overload `operator ==` , the type you declare governs how equality is defined:

- For `class` types, two objects are equal if they refer to the same object in memory.
- For `struct` types, two objects are equal if they are of the same type and store the same values.
- For `record` types, including `record struct` and `readonly record struct` , two objects are equal if they are of the same type and store the same values.

The definition of equality for a `record struct` is the same as for a `struct` . The difference is that for a `struct` , the implementation is in `ValueType.Equals(Object)` and relies on reflection. For records, the implementation is compiler synthesized and uses the declared data members.

Reference equality is required for some data models. For example, [Entity Framework Core](#) depends on reference equality to ensure that it uses only one instance of an entity type for what is conceptually one entity. For this reason, records and record structs aren't appropriate for use as entity types in Entity Framework Core.

The following example illustrates value equality of record types:

```
public record Person(string FirstName, string LastName, string[] PhoneNumbers);

public static void Main()
{
    var phoneNumbers = new string[2];
    Person person1 = new("Nancy", "Davolio", phoneNumbers);
    Person person2 = new("Nancy", "Davolio", phoneNumbers);
    Console.WriteLine(person1 == person2); // output: True

    person1.PhoneNumbers[0] = "555-1234";
    Console.WriteLine(person1 == person2); // output: True

    Console.WriteLine(ReferenceEquals(person1, person2)); // output: False
}
```

To implement value equality, the compiler synthesizes the following methods:

- An override of `Object.Equals(Object)`.

This method is used as the basis for the `Object.Equals(Object, Object)` static method when both parameters are non-null.

- A virtual `Equals` method whose parameter is the record type. This method implements `IEquatable<T>`.
- An override of `Object.GetHashCode()`.
- Overrides of operators `==` and `!=`.

You can write your own implementations to replace any of these synthesized methods. If a record type has a method that matches the signature of any synthesized method, the compiler doesn't synthesize that method.

If you provide your own implementation of `Equals` in a record type, provide an implementation of `GetHashCode` also.

Nondestructive mutation

If you need to copy an instance with some modifications, you can use a `with` expression to achieve *nondestructive mutation*. A `with` expression makes a new record instance that is a copy of an existing record instance, with specified properties and fields modified. You use [object initializer](#) syntax to specify the values to be changed, as shown in the following example:

```
public record Person(string FirstName, string LastName)
{
    public string[] PhoneNumbers { get; init; }
}

public static void Main()
{
    Person person1 = new("Nancy", "Davolio") { PhoneNumbers = new string[1] };
    Console.WriteLine(person1);
    // output: Person { FirstName = Nancy, LastName = Davolio, PhoneNumbers = System.String[] }

    Person person2 = person1 with { FirstName = "John" };
    Console.WriteLine(person2);
    // output: Person { FirstName = John, LastName = Davolio, PhoneNumbers = System.String[] }
    Console.WriteLine(person1 == person2); // output: False

    person2 = person1 with { PhoneNumbers = new string[1] };
    Console.WriteLine(person2);
    // output: Person { FirstName = Nancy, LastName = Davolio, PhoneNumbers = System.String[] }
    Console.WriteLine(person1 == person2); // output: False

    person2 = person1 with { };
    Console.WriteLine(person1 == person2); // output: True
}
```

The `with` expression can set positional properties or properties created by using standard property syntax. Non-positional properties must have an `init` or `set` accessor to be changed in a `with` expression.

The result of a `with` expression is a *shallow copy*, which means that for a reference property, only the reference to an instance is copied. Both the original record and the copy end up with a reference to the same instance.

To implement this feature for `record class` types, the compiler synthesizes a clone method and a copy constructor. The virtual clone method returns a new record initialized by the copy constructor. When you use a `with` expression, the compiler creates code that calls the clone method and then sets the properties that are specified in the `with` expression.

If you need different copying behavior, you can write your own copy constructor in a `record class`. If you do

that, the compiler won't synthesize one. Make your constructor `private` if the record is `sealed`, otherwise make it `protected`. The compiler doesn't synthesize a copy constructor for `record struct` types. You can write one, but the compiler won't generate calls to it for `with` expressions. The values of the `record struct` are copied on assignment.

You can't override the clone method, and you can't create a member named `clone` in any record type. The actual name of the clone method is compiler-generated.

Built-in formatting for display

Record types have a compiler-generated `ToString` method that displays the names and values of public properties and fields. The `ToString` method returns a string of the following format:

```
<record type name> { <property name> = <value>, <property name> = <value>, ...}
```

The string printed for `<value>` is the string returned by the `ToString()` for the type of the property. In the following example, `ChildNames` is a `System.Array`, where `ToString` returns `System.String[]`:

```
Person { FirstName = Nancy, LastName = Davolio, ChildNames = System.String[] }
```

To implement this feature, in `record class` types, the compiler synthesizes a virtual `PrintMembers` method and a `ToString` override. In `record struct` types, this member is `private`. The `ToString` override creates a `StringBuilder` object with the type name followed by an opening bracket. It calls `PrintMembers` to add property names and values, then adds the closing bracket. The following example shows code similar to what the synthesized override contains:

```
public override string ToString()
{
    StringBuilder stringBuilder = new StringBuilder();
    stringBuilder.Append("Teacher"); // type name
    stringBuilder.Append(" { ");
    if (PrintMembers(stringBuilder))
    {
        stringBuilder.Append(" ");
    }
    stringBuilder.Append("}");
    return stringBuilder.ToString();
}
```

You can provide your own implementation of `PrintMembers` or the `ToString` override. Examples are provided in the [PrintMembers formatting in derived records](#) section later in this article. In C# 10 and later, your implementation of `ToString` may include the `sealed` modifier, which prevents the compiler from synthesizing a `ToString` implementation for any derived records. You can do this to create a consistent string representation throughout a hierarchy of `record` types. (Derived records will still have a `PrintMembers` method generated for all derived properties.)

Inheritance

This section only applies to `record class` types.

A record can inherit from another record. However, a record can't inherit from a class, and a class can't inherit from a record.

Positional parameters in derived record types

The derived record declares positional parameters for all the parameters in the base record primary constructor.

The base record declares and initializes those properties. The derived record doesn't hide them, but only creates and initializes properties for parameters that aren't declared in its base record.

The following example illustrates inheritance with positional property syntax:

```
public abstract record Person(string FirstName, string LastName);
public record Teacher(string FirstName, string LastName, int Grade)
    : Person(FirstName, LastName);
public static void Main()
{
    Person teacher = new Teacher("Nancy", "Davolio", 3);
    Console.WriteLine(teacher);
    // output: Teacher { FirstName = Nancy, LastName = Davolio, Grade = 3 }
}
```

Equality in inheritance hierarchies

This section applies to `record class` types, but not `record struct` types. For two record variables to be equal, the run-time type must be equal. The types of the containing variables might be different. Inherited equality comparison is illustrated in the following code example:

```
public abstract record Person(string FirstName, string LastName);
public record Teacher(string FirstName, string LastName, int Grade)
    : Person(FirstName, LastName);
public record Student(string FirstName, string LastName, int Grade)
    : Person(FirstName, LastName);
public static void Main()
{
    Person teacher = new Teacher("Nancy", "Davolio", 3);
    Person student = new Student("Nancy", "Davolio", 3);
    Console.WriteLine(teacher == student); // output: False

    Student student2 = new Student("Nancy", "Davolio", 3);
    Console.WriteLine(student2 == student); // output: True
}
```

In the example, all variables are declared as `Person`, even when the instance is a derived type of either `Student` or `Teacher`. The instances have the same properties and the same property values. But `student == teacher` returns `False` although both are `Person`-type variables, and `student == student2` returns `True` although one is a `Person` variable and one is a `Student` variable. The equality test depends on the runtime type of the actual object, not the declared type of the variable.

To implement this behavior, the compiler synthesizes an `EqualityContract` property that returns a `Type` object that matches the type of the record. The `EqualityContract` enables the equality methods to compare the runtime type of objects when they're checking for equality. If the base type of a record is `object`, this property is `virtual`. If the base type is another record type, this property is an override. If the record type is `sealed`, this property is effectively `sealed` because the type is `sealed`.

When comparing two instances of a derived type, the synthesized equality methods check all properties of the base and derived types for equality. The synthesized `GetHashCode` method uses the `GetHashCode` method from all properties and fields declared in the base type and the derived record type.

with expressions in derived records

The result of a `with` expression has the same run-time type as the expression's operand. All properties of the run-time type get copied, but you can only set properties of the compile-time type, as the following example shows:

```

public record Point(int X, int Y)
{
    public int Zbase { get; set; }
};
public record NamedPoint(string Name, int X, int Y) : Point(X, Y)
{
    public int Zderived { get; set; }
};

public static void Main()
{
    Point p1 = new NamedPoint("A", 1, 2) { Zbase = 3, Zderived = 4 };

    Point p2 = p1 with { X = 5, Y = 6, Zbase = 7 }; // Can't set Name or Zderived
    Console.WriteLine(p2 is NamedPoint); // output: True
    Console.WriteLine(p2);
    // output: NamedPoint { X = 5, Y = 6, Zbase = 7, Name = A, Zderived = 4 }

    Point p3 = (NamedPoint)p1 with { Name = "B", X = 5, Y = 6, Zbase = 7, Zderived = 8 };
    Console.WriteLine(p3);
    // output: NamedPoint { X = 5, Y = 6, Zbase = 7, Name = B, Zderived = 8 }
}

```

PrintMembers **formatting in derived records**

The synthesized `PrintMembers` method of a derived record type calls the base implementation. The result is that all public properties and fields of both derived and base types are included in the `ToString` output, as shown in the following example:

```

public abstract record Person(string FirstName, string LastName);
public record Teacher(string FirstName, string LastName, int Grade)
    : Person(FirstName, LastName);
public record Student(string FirstName, string LastName, int Grade)
    : Person(FirstName, LastName);

public static void Main()
{
    Person teacher = new Teacher("Nancy", "Davolio", 3);
    Console.WriteLine(teacher);
    // output: Teacher { FirstName = Nancy, LastName = Davolio, Grade = 3 }
}

```

You can provide your own implementation of the `PrintMembers` method. If you do that, use the following signature:

- For a `sealed` record that derives from `object` (doesn't declare a base record):
`private bool PrintMembers(StringBuilder builder);`
- For a `sealed` record that derives from another record (note that the enclosing type is `sealed`, so the method is effectively `sealed`): `protected override bool PrintMembers(StringBuilder builder);`
- For a record that isn't `sealed` and derives from `object`:
`protected virtual bool PrintMembers(StringBuilder builder);`
- For a record that isn't `sealed` and derives from another record:
`protected override bool PrintMembers(StringBuilder builder);`

Here's an example of code that replaces the synthesized `PrintMembers` methods, one for a record type that derives from `object`, and one for a record type that derives from another record:

```

public abstract record Person(string FirstName, string LastName, string[] PhoneNumbers)
{
    protected virtual bool PrintMembers(StringBuilder stringBuilder)
    {
        stringBuilder.Append($"FirstName = {FirstName}, LastName = {LastName}, ");
        stringBuilder.Append($"PhoneNumber1 = {PhoneNumbers[0]}, PhoneNumber2 = {PhoneNumbers[1]}");
        return true;
    }
}

public record Teacher(string FirstName, string LastName, string[] PhoneNumbers, int Grade)
    : Person(FirstName, LastName, PhoneNumbers)
{
    protected override bool PrintMembers(StringBuilder stringBuilder)
    {
        if (base.PrintMembers(stringBuilder))
        {
            stringBuilder.Append(", ");
        };
        stringBuilder.Append($"Grade = {Grade}");
        return true;
    }
};

public static void Main()
{
    Person teacher = new Teacher("Nancy", "Davolio", new string[2] { "555-1234", "555-6789" }, 3);
    Console.WriteLine(teacher);
    // output: Teacher { FirstName = Nancy, LastName = Davolio, PhoneNumber1 = 555-1234, PhoneNumber2 = 555-
    6789, Grade = 3 }
}

```

NOTE

In C# 10 and later, the compiler will synthesize `PrintMembers` in derived records even when a base record has sealed the `ToString` method. You can also create your own implementation of `PrintMembers`.

Deconstructor behavior in derived records

The `Deconstruct` method of a derived record returns the values of all positional properties of the compile-time type. If the variable type is a base record, only the base record properties are deconstructed unless the object is cast to the derived type. The following example demonstrates calling a deconstructor on a derived record.

```

public abstract record Person(string FirstName, string LastName);
public record Teacher(string FirstName, string LastName, int Grade)
    : Person(FirstName, LastName);
public record Student(string FirstName, string LastName, int Grade)
    : Person(FirstName, LastName);

public static void Main()
{
    Person teacher = new Teacher("Nancy", "Davolio", 3);
    var (firstName, lastName) = teacher; // Doesn't deconstruct Grade
    Console.WriteLine($"{firstName}, {lastName}");// output: Nancy, Davolio

    var (fName, lName, grade) = (Teacher)teacher;
    Console.WriteLine($"{fName}, {lName}, {grade}");// output: Nancy, Davolio, 3
}

```

Generic constraints

There's no generic constraint that requires a type to be a record. Records satisfy either the `class` or `struct` constraint. To make a constraint on a specific hierarchy of record types, put the constraint on the base record as you would a base class. For more information, see [Constraints on type parameters](#).

C# language specification

For more information, see the [Classes](#) section of the [C# language specification](#).

For more information about features introduced in C# 9 and later, see the following feature proposal notes:

- [Records](#)
- [Init-only setters](#)
- [Covariant returns](#)

See also

- [C# reference](#)
- [Design guidelines - Choosing between class and struct](#)
- [Design guidelines - Struct design](#)
- [The C# type system](#)
- `with` expression

class (C# Reference)

12/28/2021 • 2 minutes to read • [Edit Online](#)

Classes are declared using the keyword `class`, as shown in the following example:

```
class TestClass
{
    // Methods, properties, fields, events, delegates
    // and nested classes go here.
}
```

Remarks

Only single inheritance is allowed in C#. In other words, a class can inherit implementation from one base class only. However, a class can implement more than one interface. The following table shows examples of class inheritance and interface implementation:

INHERITANCE	EXAMPLE
None	<code>class ClassA { }</code>
Single	<code>class DerivedClass : BaseClass { }</code>
None, implements two interfaces	<code>class ImplClass : IFace1, IFace2 { }</code>
Single, implements one interface	<code>class ImplDerivedClass : BaseClass, IFace1 { }</code>

Classes that you declare directly within a namespace, not nested within other classes, can be either [public](#) or [internal](#). Classes are `internal` by default.

Class members, including nested classes, can be [public](#), [protected internal](#), [protected](#), [internal](#), [private](#), or [private protected](#). Members are `private` by default.

For more information, see [Access Modifiers](#).

You can declare generic classes that have type parameters. For more information, see [Generic Classes](#).

A class can contain declarations of the following members:

- [Constructors](#)
- [Constants](#)
- [Fields](#)
- [Finalizers](#)
- [Methods](#)
- [Properties](#)
- [Indexers](#)
- [Operators](#)

- [Events](#)
- [Delegates](#)
- [Classes](#)
- [Interfaces](#)
- [Structure types](#)
- [Enumeration types](#)

Example

The following example demonstrates declaring class fields, constructors, and methods. It also demonstrates object instantiation and printing instance data. In this example, two classes are declared. The first class, `Child`, contains two private fields (`name` and `age`), two public constructors and one public method. The second class, `StringTest`, is used to contain `Main`.

```

class Child
{
    private int age;
    private string name;

    // Default constructor:
    public Child()
    {
        name = "N/A";
    }

    // Constructor:
    public Child(string name, int age)
    {
        this.name = name;
        this.age = age;
    }

    // Printing method:
    public void PrintChild()
    {
        Console.WriteLine("{0}, {1} years old.", name, age);
    }
}

class StringTest
{
    static void Main()
    {
        // Create objects by using the new operator:
        Child child1 = new Child("Craig", 11);
        Child child2 = new Child("Sally", 10);

        // Create an object using the default constructor:
        Child child3 = new Child();

        // Display results:
        Console.Write("Child #1: ");
        child1.PrintChild();
        Console.Write("Child #2: ");
        child2.PrintChild();
        Console.Write("Child #3: ");
        child3.PrintChild();
    }
}
/* Output:
    Child #1: Craig, 11 years old.
    Child #2: Sally, 10 years old.
    Child #3: N/A, 0 years old.
*/

```

Comments

Notice that in the previous example the private fields (`name` and `age`) can only be accessed through the public method of the `Child` class. For example, you cannot print the child's name, from the `Main` method, using a statement like this:

```
Console.Write(child1.name);    // Error
```

Accessing private members of `Child` from `Main` would only be possible if `Main` were a member of the class.

Types declared inside a class without an access modifier default to `private`, so the data members in this

example would still be `private` if the keyword were removed.

Finally, notice that for the object created using the parameterless constructor (`child3`), the `age` field was initialized to zero by default.

C# language specification

For more information, see the [C# Language Specification](#). The language specification is the definitive source for C# syntax and usage.

See also

- [C# Reference](#)
- [C# Programming Guide](#)
- [C# Keywords](#)
- [Reference Types](#)

interface (C# Reference)

12/28/2021 • 3 minutes to read • [Edit Online](#)

An interface defines a contract. Any `class` or `struct` that implements that contract must provide an implementation of the members defined in the interface. Beginning with C# 8.0, an interface may define a default implementation for members. It may also define `static` members in order to provide a single implementation for common functionality.

In the following example, class `ImplementationClass` must implement a method named `SampleMethod` that has no parameters and returns `void`.

For more information and examples, see [Interfaces](#).

Example interface

```
interface ISampleInterface
{
    void SampleMethod();
}

class ImplementationClass : ISampleInterface
{
    // Explicit interface member implementation:
    void ISampleInterface.SampleMethod()
    {
        // Method implementation.
    }

    static void Main()
    {
        // Declare an interface instance.
        ISampleInterface obj = new ImplementationClass();

        // Call the member.
        obj.SampleMethod();
    }
}
```

An interface can be a member of a namespace or a class. An interface declaration can contain declarations (signatures without any implementation) of the following members:

- [Methods](#)
- [Properties](#)
- [Indexers](#)
- [Events](#)

These preceding member declarations typically do not contain a body. Beginning with C# 8.0, an interface member may declare a body. This is called a *default implementation*. Members with bodies permit the interface to provide a "default" implementation for classes and structs that don't provide an overriding implementation. In addition, beginning with C# 8.0, an interface may include:

- [Constants](#)
- [Operators](#)
- [Static constructor](#).

- [Nested types](#)
- [Static fields, methods, properties, indexers, and events](#)
- Member declarations using the explicit interface implementation syntax.
- Explicit access modifiers (the default access is `public`).

Interfaces may not contain instance state. While static fields are now permitted, instance fields are not permitted in interfaces. [Instance auto-properties](#) are not supported in interfaces, as they would implicitly declare a hidden field. This rule has a subtle effect on property declarations. In an interface declaration, the following code does not declare an auto-implemented property as it does in a `class` or `struct`. Instead, it declares a property that doesn't have a default implementation but must be implemented in any type that implements the interface:

```
public interface INamed
{
    public string Name {get; set;}
}
```

An interface can inherit from one or more base interfaces. When an interface [overrides a method](#) implemented in a base interface, it must use the [explicit interface implementation](#) syntax.

When a base type list contains a base class and interfaces, the base class must come first in the list.

A class that implements an interface can explicitly implement members of that interface. An explicitly implemented member cannot be accessed through a class instance, but only through an instance of the interface. In addition, default interface members can only be accessed through an instance of the interface.

For more information about explicit interface implementation, see [Explicit Interface Implementation](#).

Example interface implementation

The following example demonstrates interface implementation. In this example, the interface contains the property declaration and the class contains the implementation. Any instance of a class that implements `IPoint` has integer properties `x` and `y`.

```

interface IPoint
{
    // Property signatures:
    int X
    {
        get;
        set;
    }

    int Y
    {
        get;
        set;
    }

    double Distance
    {
        get;
    }
}

class Point : IPoint
{
    // Constructor:
    public Point(int x, int y)
    {
        X = x;
        Y = y;
    }

    // Property implementation:
    public int X { get; set; }

    public int Y { get; set; }

    // Property implementation
    public double Distance =>
        Math.Sqrt(X * X + Y * Y);
}

class MainClass
{
    static void PrintPoint(IPoint p)
    {
        Console.WriteLine("x={0}, y={1}", p.X, p.Y);
    }

    static void Main()
    {
        IPoint p = new Point(2, 3);
        Console.Write("My Point: ");
        PrintPoint(p);
    }
}
// Output: My Point: x=2, y=3

```

C# language specification

For more information, see the [Interfaces](#) section of the [C# language specification](#) and the feature specification for [Default interface members - C# 8.0](#)

See also

- [C# Reference](#)
- [C# Programming Guide](#)
- [C# Keywords](#)
- [Reference Types](#)
- [Interfaces](#)
- [Using Properties](#)
- [Using Indexers](#)

Nullable reference types (C# reference)

12/28/2021 • 5 minutes to read • [Edit Online](#)

NOTE

This article covers nullable reference types. You can also declare [nullable value types](#).

Nullable reference types are available beginning with C# 8.0, in code that has opted in to a *nullable aware context*. Nullable reference types, the null static analysis warnings, and the [null-forgiving operator](#) are optional language features. All are turned off by default. A *nullable context* is controlled at the project level using build settings, or in code using pragmas.

In a nullable aware context:

- A variable of a reference type `T` must be initialized with non-null, and may never be assigned a value that may be `null`.
- A variable of a reference type `T?` may be initialized with `null` or assigned `null`, but is required to be checked against `null` before de-referencing.
- A variable `m` of type `T?` is considered to be non-null when you apply the null-forgiving operator, as in `m!`.

The distinctions between a non-nullable reference type `T` and a nullable reference type `T?` are enforced by the compiler's interpretation of the preceding rules. A variable of type `T` and a variable of type `T?` are represented by the same .NET type. The following example declares a non-nullable string and a nullable string, and then uses the null-forgiving operator to assign a value to a non-nullable string:

```
string notNull = "Hello";
string? nullable = default;
notNull = nullable!; // null forgiveness
```

The variables `notNull` and `nullable` are both represented by the [String](#) type. Because the non-nullable and nullable types are both stored as the same type, there are several locations where using a nullable reference type isn't allowed. In general, a nullable reference type can't be used as a base class or implemented interface. A nullable reference type can't be used in any object creation or type testing expression. A nullable reference type can't be the type of a member access expression. The following examples show these constructs:

```
public MyClass : System.Object? // not allowed
{
}

var nullEmpty = System.String?.Empty; // Not allowed
var maybeObject = new object?(); // Not allowed
try
{
    if (thing is string? nullableString) // not allowed
        Console.WriteLine(nullableString);
} catch (Exception? e) // Not Allowed
{
    Console.WriteLine("error");
}
```


Nullable references and static analysis

The examples in the previous section illustrate the nature of nullable reference types. Nullable reference types aren't new class types, but rather annotations on existing reference types. The compiler uses those annotations to help you find potential null reference errors in your code. There's no runtime difference between a non-nullable reference type and a nullable reference type. The compiler doesn't add any runtime checking for non-nullable reference types. The benefits are in the compile-time analysis. The compiler generates warnings that help you find and fix potential null errors in your code. You declare your intent, and the compiler warns you when your code violates that intent.

In a nullable enabled context, the compiler performs static analysis on variables of any reference type, both nullable and non-nullable. The compiler tracks the *null-state* of each reference variable as either *not-null* or *maybe-null*. The default state of a non-nullable reference is *not-null*. The default state of a nullable reference is *maybe-null*.

Non-nullable reference types should always be safe to dereference because their *null-state* is *not-null*. To enforce that rule, the compiler issues warnings if a non-nullable reference type isn't initialized to a non-null value. Local variables must be assigned where they're declared. Every field must be assigned a *not-null* value, in a field initializer or every constructor. The compiler issues warnings when a non-nullable reference is assigned to a reference whose state is *maybe-null*. Generally, a non-nullable reference is *not-null* and no warnings are issued when those variables are dereferenced.

NOTE

If you assign a *maybe-null* expression to a non-nullable reference type, the compiler generates a warnings. The compiler then generates warnings for that variable until it's assigned to a *not-null* expression.

Nullable reference types may be initialized or assigned to `null`. Therefore, static analysis must determine that a variable is *not-null* before it's dereferenced. If a nullable reference is determined to be *maybe-null*, assigning to a non-nullable reference variable generates a compiler warning. The following class shows examples of these warnings:

```

public class ProductDescription
{
    private string shortDescription;
    private string? detailedDescription;

    public ProductDescription() // Warning! shortDescription not initialized.
    {
    }

    public ProductDescription(string productDescription) =>
        this.shortDescription = productDescription;

    public void SetDescriptions(string productDescription, string? details=null)
    {
        shortDescription = productDescription;
        detailedDescription = details;
    }

    public string GetDescription()
    {
        if (detailedDescription.Length == 0) // Warning! dereference possible null
        {
            return shortDescription;
        }
        else
        {
            return $"{shortDescription}\n{detailedDescription}";
        }
    }

    public string FullDescription()
    {
        if (detailedDescription == null)
        {
            return shortDescription;
        }
        else if (detailedDescription.Length > 0) // OK, detailedDescription can't be null.
        {
            return $"{shortDescription}\n{detailedDescription}";
        }
        return shortDescription;
    }
}

```

The following snippet shows where the compiler emits warnings when using this class:

```

string shortDescription = default; // Warning! non-nullable set to null;
var product = new ProductDescription(shortDescription); // Warning! static analysis knows shortDescription
maybe null.

string description = "widget";
var item = new ProductDescription(description);

item.SetDescriptions(description, "These widgets will do everything.");

```

The preceding examples demonstrate how compiler's static analysis determines the *null-state* of reference variables. The compiler applies language rules for null checks and assignments to inform its analysis. The compiler can't make assumptions about the semantics of methods or properties. If you call methods that perform null checks, the compiler can't know those methods affect a variable's *null-state*. There are attributes you can add to your APIs to inform the compiler about the semantics of arguments and return values. These attributes have been applied to many common APIs in the .NET Core libraries. For example, [IsNullOrEmpty](#) has been updated, and the compiler correctly interprets that method as a null check. For more information about the

attributes that apply to *null-state* static analysis, see the article on [Nullable attributes](#).

Setting the nullable context

There are two ways to control the nullable context. At the project level, you can add the `<Nullable>enable</Nullable>` project setting. In a single C# source file, you can add the `#nullable enable` pragma to enable the nullable context. See the article on [setting a nullable strategy](#). Prior to .NET 6, new projects use the default, `<Nullable>disable</Nullable>`. Beginning with .NET 6, new projects include the `<Nullable>enable</Nullable>` element in the project file.

C# language specification

For more information, see the following proposals for the [C# language specification](#):

- [Nullable reference types](#)
- [Draft nullable reference types specification](#)

See also

- [C# reference](#)
- [Nullable value types](#)

void (C# reference)

12/28/2021 • 2 minutes to read • [Edit Online](#)

You use `void` as the return type of a [method](#) (or a [local function](#)) to specify that the method doesn't return a value.

```
public static void Display(IEnumerable<int> numbers)
{
    if (numbers is null)
    {
        return;
    }

    Console.WriteLine(string.Join(" ", numbers));
}
```

You can also use `void` as a referent type to declare a pointer to an unknown type. For more information, see [Pointer types](#).

You cannot use `void` as the type of a variable.

See also

- [C# reference](#)
- [System.Void](#)

var (C# reference)

12/28/2021 • 2 minutes to read • [Edit Online](#)

Beginning with C# 3, variables that are declared at method scope can have an implicit "type" `var`. An implicitly typed local variable is strongly typed just as if you had declared the type yourself, but the compiler determines the type. The following two declarations of `i` are functionally equivalent:

```
var i = 10; // Implicitly typed.  
int i = 10; // Explicitly typed.
```

IMPORTANT

When `var` is used with [nullable reference types](#) enabled, it always implies a nullable reference type even if the expression type isn't nullable. The compiler's null state analysis protects against dereferencing a potential `null` value. If the variable is never assigned to an expression that maybe null, the compiler won't emit any warnings. If you assign the variable to an expression that might be null, you must test that it isn't null before dereferencing it to avoid any warnings.

A common use of the `var` keyword is with constructor invocation expressions. The use of `var` allows you to not repeat a type name in a variable declaration and object instantiation, as the following example shows:

```
var xs = new List<int>();
```

Beginning with C# 9.0, you can use a target-typed `new` expression as an alternative:

```
List<int> xs = new();  
List<int>? ys = new();
```

In pattern matching, the `var` keyword is used in a `var` pattern.

Example

The following example shows two query expressions. In the first expression, the use of `var` is permitted but is not required, because the type of the query result can be stated explicitly as an `IEnumerable<string>`. However, in the second expression, `var` allows the result to be a collection of anonymous types, and the name of that type is not accessible except to the compiler itself. Use of `var` eliminates the requirement to create a new class for the result. Note that in Example #2, the `foreach` iteration variable `item` must also be implicitly typed.

```

// Example #1: var is optional when
// the select clause specifies a string
string[] words = { "apple", "strawberry", "grape", "peach", "banana" };
var wordQuery = from word in words
                 where word[0] == 'g'
                 select word;

// Because each element in the sequence is a string,
// not an anonymous type, var is optional here also.
foreach (string s in wordQuery)
{
    Console.WriteLine(s);
}

// Example #2: var is required because
// the select clause specifies an anonymous type
var custQuery = from cust in customers
                 where cust.City == "Phoenix"
                 select new { cust.Name, cust.Phone };

// var must be used because each item
// in the sequence is an anonymous type
foreach (var item in custQuery)
{
    Console.WriteLine("Name={0}, Phone={1}", item.Name, item.Phone);
}

```

See also

- [C# reference](#)
- [Implicitly typed local variables](#)
- [Type relationships in LINQ query operations](#)

Built-in types (C# reference)

12/28/2021 • 2 minutes to read • [Edit Online](#)

The following table lists the C# built-in [value](#) types:

C# TYPE KEYWORD	.NET TYPE
<code>bool</code>	<code>System.Boolean</code>
<code>byte</code>	<code>System.Byte</code>
<code>sbyte</code>	<code>System.SByte</code>
<code>char</code>	<code>System.Char</code>
<code>decimal</code>	<code>System.Decimal</code>
<code>double</code>	<code>System.Double</code>
<code>float</code>	<code>System.Single</code>
<code>int</code>	<code>System.Int32</code>
<code>uint</code>	<code>System.UInt32</code>
<code>nint</code>	<code>System.IntPtr</code>
<code>nuint</code>	<code>System.UIntPtr</code>
<code>long</code>	<code>System.Int64</code>
<code>ulong</code>	<code>System.UInt64</code>
<code>short</code>	<code>System.Int16</code>
<code>ushort</code>	<code>System.UInt16</code>

The following table lists the C# built-in [reference](#) types:

C# TYPE KEYWORD	.NET TYPE
<code>object</code>	<code>System.Object</code>
<code>string</code>	<code>System.String</code>
<code>dynamic</code>	<code>System.Object</code>

In the preceding tables, each C# type keyword from the left column (except [nint](#) and [nuint](#) and [dynamic](#)) is an

alias for the corresponding .NET type. They are interchangeable. For example, the following declarations declare variables of the same type:

```
int a = 123;  
System.Int32 b = 123;
```

The `int` and `uint` types are native-sized integers. They are represented internally by the indicated .NET types, but in each case the keyword and the .NET type are not interchangeable. The compiler provides operations and conversions for `int` and `uint` as integer types that it doesn't provide for the pointer types `System.IntPtr` and `System.UIntPtr`. For more information, see [int and uint types](#).

The `void` keyword represents the absence of a type. You use it as the return type of a method that doesn't return a value.

See also

- [C# reference](#)
- [Default values of C# types](#)

Unmanaged types (C# reference)

12/28/2021 • 2 minutes to read • [Edit Online](#)

A type is an **unmanaged type** if it's any of the following types:

- `sbyte`, `byte`, `short`, `ushort`, `int`, `uint`, `long`, `ulong`, `char`, `float`, `double`, `decimal`, or `bool`
- Any **enum** type
- Any **pointer** type
- Any user-defined **struct** type that contains fields of unmanaged types only and, in C# 7.3 and earlier, is not a constructed type (a type that includes at least one type argument)

Beginning with C# 7.3, you can use the **unmanaged** **constraint** to specify that a type parameter is a non-pointer, non-nullable unmanaged type.

Beginning with C# 8.0, a *constructed* struct type that contains fields of unmanaged types only is also unmanaged, as the following example shows:

```
using System;

public struct Coords<T>
{
    public T X;
    public T Y;
}

public class UnmanagedType
{
    public static void Main()
    {
        DisplaySize<Coords<int>>();
        DisplaySize<Coords<double>>();
    }

    private unsafe static void DisplaySize<T>() where T : unmanaged
    {
        Console.WriteLine($"{typeof(T)} is unmanaged and its size is {sizeof(T)} bytes");
    }
}

// Output:
// Coords`1[System.Int32] is unmanaged and its size is 8 bytes
// Coords`1[System.Double] is unmanaged and its size is 16 bytes
```

A generic struct may be the source of both unmanaged and not unmanaged constructed types. The preceding example defines a generic struct `Coords<T>` and presents the examples of unmanaged constructed types. The example of not an unmanaged type is `Coords<object>`. It's not unmanaged because it has the fields of the `object` type, which is not unmanaged. If you want *all* constructed types to be unmanaged types, use the **unmanaged** constraint in the definition of a generic struct:

```
public struct Coords<T> where T : unmanaged
{
    public T X;
    public T Y;
}
```

C# language specification

For more information, see the [Pointer types](#) section of the [C# language specification](#).

See also

- [C# reference](#)
- [Pointer types](#)
- [Memory and span-related types](#)
- [sizeof operator](#)
- [stackalloc](#)

Default values of C# types (C# reference)

12/28/2021 • 2 minutes to read • [Edit Online](#)

The following table shows the default values of C# types:

TYPE	DEFAULT VALUE
Any reference type	<code>null</code>
Any built-in integral numeric type	0 (zero)
Any built-in floating-point numeric type	0 (zero)
bool	<code>false</code>
char	<code>'\0'</code> (U+0000)
enum	The value produced by the expression <code>(E)0</code> , where <code>E</code> is the enum identifier.
struct	The value produced by setting all value-type fields to their default values and all reference-type fields to <code>null</code> .
Any nullable value type	An instance for which the HasValue property is <code>false</code> and the Value property is undefined. That default value is also known as the <i>null</i> value of a nullable value type.

Default value expressions

Use the [default](#) operator to produce the default value of a type, as the following example shows:

```
int a = default(int);
```

Beginning with C# 7.1, you can use the [default](#) literal to initialize a variable with the default value of its type:

```
int a = default;
```

Parameterless constructor of a value type

For a value type, the *implicit* parameterless constructor also produces the default value of the type, as the following example shows:

```
var n = new System.Numerics.Complex();  
Console.WriteLine(n); // output: (0, 0)
```

At run time, if the [System.Type](#) instance represents a value type, you can use the [Activator.CreateInstance\(Type\)](#) method to invoke the parameterless constructor to obtain the default value of the type.

NOTE

In C# 10 and later, a [structure type](#) (which is a value type) may have an [explicit parameterless constructor](#) that may produce a non-default value of the type. Thus, we recommend using the `default` operator or the `default` literal to produce the default value of a type.

C# language specification

For more information, see the following sections of the [C# language specification](#):

- [Default values](#)
- [Default constructors](#)

See also

- [C# reference](#)
- [Constructors](#)

C# Keywords

12/28/2021 • 2 minutes to read • [Edit Online](#)

Keywords are predefined, reserved identifiers that have special meanings to the compiler. They cannot be used as identifiers in your program unless they include `@` as a prefix. For example, `@if` is a valid identifier, but `if` is not because `if` is a keyword.

The first table in this topic lists keywords that are reserved identifiers in any part of a C# program. The second table in this topic lists the contextual keywords in C#. Contextual keywords have special meaning only in a limited program context and can be used as identifiers outside that context. Generally, as new keywords are added to the C# language, they are added as contextual keywords in order to avoid breaking programs written in earlier versions.

abstract

as

base

bool

break

byte

case

catch

char

checked

class

const

continue

decimal

default

delegate

do

double

else

enum

event

explicit

extern

false

finally

fixed

float

for

foreach

goto

if

implicit

in

int

interface

internal

is
lock
long

namespace
new
null
object
operator
out
override
params
private
protected
public
readonly
ref
return
sbyte
sealed
short
sizeof
stackalloc

static
string
struct
switch
this
throw
true
try
typeof
uint
ulong
unchecked
unsafe
ushort
using
virtual
void
volatile
while

Contextual keywords

A contextual keyword is used to provide a specific meaning in the code, but it is not a reserved word in C#. Some contextual keywords, such as `partial` and `where`, have special meanings in two or more contexts.

add
and
alias
ascending

[async](#)
[await](#)
[by](#)
[descending](#)
[dynamic](#)
[equals](#)
[from](#)

[get](#)
[global](#)
[group](#)
[init](#)
[into](#)
[join](#)
[let](#)
[managed \(function pointer calling convention\)](#)
[nameof](#)
[nint](#)
[not](#)

[notnull](#)
[nuint](#)
[on](#)
[or](#)
[orderby](#)
[partial \(type\)](#)
[partial \(method\)](#)
[record](#)
[remove](#)
[select](#)

[set](#)
[unmanaged \(function pointer calling convention\)](#)
[unmanaged \(generic type constraint\)](#)
[value](#)
[var](#)
[when \(filter condition\)](#)
[where \(generic type constraint\)](#)
[where \(query clause\)](#)
[with](#)
[yield](#)

See also

- [C# reference](#)

Access Modifiers (C# Reference)

12/28/2021 • 2 minutes to read • [Edit Online](#)

Access modifiers are keywords used to specify the declared accessibility of a member or a type. This section introduces the four access modifiers:

- `public`
- `protected`
- `internal`
- `private`

The following six accessibility levels can be specified using the access modifiers:

- `public` : Access is not restricted.
- `protected` : Access is limited to the containing class or types derived from the containing class.
- `internal` : Access is limited to the current assembly.
- `protected internal` : Access is limited to the current assembly or types derived from the containing class.
- `private` : Access is limited to the containing type.
- `private protected` : Access is limited to the containing class or types derived from the containing class within the current assembly.

This section also introduces the following:

- [Accessibility Levels](#): Using the four access modifiers to declare six levels of accessibility.
- [Accessibility Domain](#): Specifies where, in the program sections, a member can be referenced.
- [Restrictions on Using Accessibility Levels](#): A summary of the restrictions on using declared accessibility levels.

See also

- [C# Reference](#)
- [C# Programming Guide](#)
- [C# Keywords](#)
- [Access Modifiers](#)
- [Access Keywords](#)
- [Modifiers](#)

Accessibility Levels (C# Reference)

12/28/2021 • 2 minutes to read • [Edit Online](#)

Use the access modifiers, `public`, `protected`, `internal`, or `private`, to specify one of the following declared accessibility levels for members.

DECLARED ACCESSIBILITY	MEANING
<code>public</code>	Access is not restricted.
<code>protected</code>	Access is limited to the containing class or types derived from the containing class.
<code>internal</code>	Access is limited to the current assembly.
<code>protected internal</code>	Access is limited to the current assembly or types derived from the containing class.
<code>private</code>	Access is limited to the containing type.
<code>private protected</code>	Access is limited to the containing class or types derived from the containing class within the current assembly. Available since C# 7.2.

Only one access modifier is allowed for a member or type, except when you use the `protected internal` or `private protected` combinations.

Access modifiers are not allowed on namespaces. Namespaces have no access restrictions.

Depending on the context in which a member declaration occurs, only certain declared accessibilities are permitted. If no access modifier is specified in a member declaration, a default accessibility is used.

Top-level types, which are not nested in other types, can only have `internal` or `public` accessibility. The default accessibility for these types is `internal`.

Nested types, which are members of other types, can have declared accessibilities as indicated in the following table.

MEMBERS OF	DEFAULT MEMBER ACCESSIBILITY	ALLOWED DECLARED ACCESSIBILITY OF THE MEMBER
<code>enum</code>	<code>public</code>	None

MEMBERS OF	DEFAULT MEMBER ACCESSIBILITY	ALLOWED DECLARED ACCESSIBILITY OF THE MEMBER
<code>class</code>	<code>private</code>	<code>public</code> <code>protected</code> <code>internal</code> <code>private</code> <code>protected internal</code> <code>private protected</code>
<code>interface</code>	<code>public</code>	<code>public</code> <code>protected</code> <code>internal</code> <code>private</code> * <code>protected internal</code> <code>private protected</code>
<code>struct</code>	<code>private</code>	<code>public</code> <code>internal</code> <code>private</code>

* An `interface` member with `private` accessibility must have a default implementation.

The accessibility of a nested type depends on its [accessibility domain](#), which is determined by both the declared accessibility of the member and the accessibility domain of the immediately containing type. However, the accessibility domain of a nested type cannot exceed that of the containing type.

C# Language Specification

For more information, see the [C# Language Specification](#). The language specification is the definitive source for C# syntax and usage.

See also

- [C# Reference](#)
- [C# Programming Guide](#)
- [C# Keywords](#)
- [Access Modifiers](#)
- [Accessibility Domain](#)
- [Restrictions on Using Accessibility Levels](#)
- [Access Modifiers](#)
- [public](#)
- [private](#)

- protected
- internal

Accessibility Domain (C# Reference)

12/28/2021 • 2 minutes to read • [Edit Online](#)

The accessibility domain of a member specifies in which program sections a member can be referenced. If the member is nested within another type, its accessibility domain is determined by both the [accessibility level](#) of the member and the accessibility domain of the immediately containing type.

The accessibility domain of a top-level type is at least the program text of the project that it is declared in. That is, the domain includes all of the source files of this project. The accessibility domain of a nested type is at least the program text of the type in which it is declared. That is, the domain is the type body, which includes all nested types. The accessibility domain of a nested type never exceeds that of the containing type. These concepts are demonstrated in the following example.

Example

This example contains a top-level type, `T1`, and two nested classes, `M1` and `M2`. The classes contain fields that have different declared accessibilities. In the `Main` method, a comment follows each statement to indicate the accessibility domain of each member. Notice that the statements that try to reference the inaccessible members are commented out. If you want to see the compiler errors caused by referencing an inaccessible member, remove the comments one at a time.

```
public class T1
{
    public static int publicInt;
    internal static int internalInt;
    private static int privateInt = 0;

    static T1()
    {
        // T1 can access public or internal members
        // in a public or private (or internal) nested class.
        M1.publicInt = 1;
        M1.internalInt = 2;
        M2.publicInt = 3;
        M2.internalInt = 4;

        // Cannot access the private member privateInt
        // in either class:
        // M1.privateInt = 2; //CS0122
    }

    public class M1
    {
        public static int publicInt;
        internal static int internalInt;
        private static int privateInt = 0;
    }

    private class M2
    {
        public static int publicInt = 0;
        internal static int internalInt = 0;
        private static int privateInt = 0;
    }
}

class MainClass
{
```

```

static void Main()
{
    // Access is unlimited.
    T1.publicInt = 1;

    // Accessible only in current assembly.
    T1.internalInt = 2;

    // Error CS0122: inaccessible outside T1.
    // T1.privateInt = 3;

    // Access is unlimited.
    T1.M1.publicInt = 1;

    // Accessible only in current assembly.
    T1.M1.internalInt = 2;

    // Error CS0122: inaccessible outside M1.
    //     T1.M1.privateInt = 3;

    // Error CS0122: inaccessible outside T1.
    //     T1.M2.publicInt = 1;

    // Error CS0122: inaccessible outside T1.
    //     T1.M2.internalInt = 2;

    // Error CS0122: inaccessible outside M2.
    //     T1.M2.privateInt = 3;

    // Keep the console open in debug mode.
    System.Console.WriteLine("Press any key to exit.");
    System.Console.ReadKey();
}
}

```

C# Language Specification

For more information, see the [C# Language Specification](#). The language specification is the definitive source for C# syntax and usage.

See also

- [C# Reference](#)
- [C# Programming Guide](#)
- [C# Keywords](#)
- [Access Modifiers](#)
- [Accessibility Levels](#)
- [Restrictions on Using Accessibility Levels](#)
- [Access Modifiers](#)
- [public](#)
- [private](#)
- [protected](#)
- [internal](#)

Restrictions on using accessibility levels (C# Reference)

12/28/2021 • 2 minutes to read • [Edit Online](#)

When you specify a type in a declaration, check whether the accessibility level of the type is dependent on the accessibility level of a member or of another type. For example, the direct base class must be at least as accessible as the derived class. The following declarations cause a compiler error because the base class

`BaseClass` is less accessible than `MyClass`:

```
class BaseClass {...}  
public class MyClass: BaseClass {...} // Error
```

The following table summarizes the restrictions on declared accessibility levels.

CONTEXT	REMARKS
Classes	The direct base class of a class type must be at least as accessible as the class type itself.
Interfaces	The explicit base interfaces of an interface type must be at least as accessible as the interface type itself.
Delegates	The return type and parameter types of a delegate type must be at least as accessible as the delegate type itself.
Constants	The type of a constant must be at least as accessible as the constant itself.
Fields	The type of a field must be at least as accessible as the field itself.
Methods	The return type and parameter types of a method must be at least as accessible as the method itself.
Properties	The type of a property must be at least as accessible as the property itself.
Events	The type of an event must be at least as accessible as the event itself.
Indexers	The type and parameter types of an indexer must be at least as accessible as the indexer itself.
Operators	The return type and parameter types of an operator must be at least as accessible as the operator itself.
Constructors	The parameter types of a constructor must be at least as accessible as the constructor itself.

Example

The following example contains erroneous declarations of different types. The comment following each declaration indicates the expected compiler error.

```
// Restrictions on Using Accessibility Levels
// CS0052 expected as well as CS0053, CS0056, and CS0057
// To make the program work, change access level of both class B
// and MyPrivateMethod() to public.

using System;

// A delegate:
delegate int MyDelegate();

class B
{
    // A private method:
    static int MyPrivateMethod()
    {
        return 0;
    }
}

public class A
{
    // Error: The type B is less accessible than the field A.myField.
    public B myField = new B();

    // Error: The type B is less accessible
    // than the constant A.myConst.
    public readonly B myConst = new B();

    public B MyMethod()
    {
        // Error: The type B is less accessible
        // than the method A.MyMethod.
        return new B();
    }

    // Error: The type B is less accessible than the property A.MyProp
    public B MyProp
    {
        set
        {
        }
    }

    MyDelegate d = new MyDelegate(B.MyPrivateMethod);
    // Even when B is declared public, you still get the error:
    // "The parameter B.MyPrivateMethod is not accessible due to
    // protection level."

    public static B operator +(A m1, B m2)
    {
        // Error: The type B is less accessible
        // than the operator A.operator +(A,B)
        return new B();
    }

    static void Main()
    {
        Console.Write("Compiled successfully");
    }
}
```

C# language specification

For more information, see the [C# Language Specification](#). The language specification is the definitive source for C# syntax and usage.

See also

- [C# Reference](#)
- [C# Programming Guide](#)
- [C# Keywords](#)
- [Access Modifiers](#)
- [Accessibility Domain](#)
- [Accessibility Levels](#)
- [Access Modifiers](#)
- [public](#)
- [private](#)
- [protected](#)
- [internal](#)

internal (C# Reference)

12/28/2021 • 2 minutes to read • [Edit Online](#)

The `internal` keyword is an [access modifier](#) for types and type members.

This page covers `internal` access. The `internal` keyword is also part of the `protected internal` access modifier.

Internal types or members are accessible only within files in the same assembly, as in this example:

```
public class BaseClass
{
    // Only accessible within the same assembly.
    internal static int x = 0;
}
```

For a comparison of `internal` with the other access modifiers, see [Accessibility Levels](#) and [Access Modifiers](#).

For more information about assemblies, see [Assemblies in .NET](#).

A common use of internal access is in component-based development because it enables a group of components to cooperate in a private manner without being exposed to the rest of the application code. For example, a framework for building graphical user interfaces could provide `Control` and `Form` classes that cooperate by using members with internal access. Since these members are internal, they are not exposed to code that is using the framework.

It is an error to reference a type or a member with internal access outside the assembly within which it was defined.

Example 1

This example contains two files, `Assembly1.cs` and `Assembly1_a.cs`. The first file contains an internal base class, `BaseClass`. In the second file, an attempt to instantiate `BaseClass` will produce an error.

```
// Assembly1.cs
// Compile with: /target:library
internal class BaseClass
{
    public static int intM = 0;
}
```

```
// Assembly1_a.cs
// Compile with: /reference:Assembly1.dll
class TestAccess
{
    static void Main()
    {
        var myBase = new BaseClass(); // CS0122
    }
}
```

Example 2

In this example, use the same files you used in example 1, and change the accessibility level of `BaseClass` to `public`. Also change the accessibility level of the member `intM` to `internal`. In this case, you can instantiate the class, but you cannot access the internal member.

```
// Assembly2.cs
// Compile with: /target:library
public class BaseClass
{
    internal static int intM = 0;
}
```

```
// Assembly2_a.cs
// Compile with: /reference:Assembly2.dll
public class TestAccess
{
    static void Main()
    {
        var myBase = new BaseClass();    // Ok.
        BaseClass.intM = 444;           // CS0117
    }
}
```

C# Language Specification

For more information, see [Declared accessibility](#) in the [C# Language Specification](#). The language specification is the definitive source for C# syntax and usage.

See also

- [C# Reference](#)
- [C# Programming Guide](#)
- [C# Keywords](#)
- [Access Modifiers](#)
- [Accessibility Levels](#)
- [Modifiers](#)
- [public](#)
- [private](#)
- [protected](#)

private (C# Reference)

12/28/2021 • 2 minutes to read • [Edit Online](#)

The `private` keyword is a member access modifier.

This page covers `private` access. The `private` keyword is also part of the `private protected` access modifier.

Private access is the least permissive access level. Private members are accessible only within the body of the class or the struct in which they are declared, as in this example:

```
class Employee
{
    private int _i;
    double _d;    // private access by default
}
```

Nested types in the same body can also access those private members.

It is a compile-time error to reference a private member outside the class or the struct in which it is declared.

For a comparison of `private` with the other access modifiers, see [Accessibility Levels](#) and [Access Modifiers](#).

Example

In this example, the `Employee` class contains two private data members, `_name` and `_salary`. As private members, they cannot be accessed except by member methods. Public methods named `GetName` and `Salary` are added to allow controlled access to the private members. The `_name` member is accessed by way of a public method, and the `_salary` member is accessed by way of a public read-only property. (See [Properties](#) for more information.)

```

class Employee2
{
    private readonly string _name = "FirstName, LastName";
    private readonly double _salary = 100.0;

    public string GetName()
    {
        return _name;
    }

    public double Salary
    {
        get { return _salary; }
    }
}

class PrivateTest
{
    static void Main()
    {
        var e = new Employee2();

        // The data members are inaccessible (private), so
        // they can't be accessed like this:
        //     string n = e._name;
        //     double s = e._salary;

        // '_name' is indirectly accessed via method:
        string n = e.GetName();

        // '_salary' is indirectly accessed via property
        double s = e.Salary;
    }
}

```

C# language specification

For more information, see [Declared accessibility](#) in the [C# Language Specification](#). The language specification is the definitive source for C# syntax and usage.

See also

- [C# Reference](#)
- [C# Programming Guide](#)
- [C# Keywords](#)
- [Access Modifiers](#)
- [Accessibility Levels](#)
- [Modifiers](#)
- [public](#)
- [protected](#)
- [internal](#)

protected (C# Reference)

12/28/2021 • 2 minutes to read • [Edit Online](#)

The `protected` keyword is a member access modifier.

NOTE

This page covers `protected` access. The `protected` keyword is also part of the `protected internal` and `private protected` access modifiers.

A protected member is accessible within its class and by derived class instances.

For a comparison of `protected` with the other access modifiers, see [Accessibility Levels](#).

Example 1

A protected member of a base class is accessible in a derived class only if the access occurs through the derived class type. For example, consider the following code segment:

```
class A
{
    protected int x = 123;
}

class B : A
{
    static void Main()
    {
        var a = new A();
        var b = new B();

        // Error CS1540, because x can only be accessed by
        // classes derived from A.
        // a.x = 10;

        // OK, because this class derives from A.
        b.x = 10;
    }
}
```

The statement `a.x = 10` generates an error because it is made within the static method `Main`, and not an instance of class `B`.

Struct members cannot be protected because the struct cannot be inherited.

Example 2

In this example, the class `DerivedPoint` is derived from `Point`. Therefore, you can access the protected members of the base class directly from the derived class.

```

class Point
{
    protected int x;
    protected int y;
}

class DerivedPoint: Point
{
    static void Main()
    {
        var dpoint = new DerivedPoint();

        // Direct access to protected members.
        dpoint.x = 10;
        dpoint.y = 15;
        Console.WriteLine($"x = {dpoint.x}, y = {dpoint.y}");
    }
}
// Output: x = 10, y = 15

```

If you change the access levels of `x` and `y` to [private](#), the compiler will issue the error messages:

```
'Point.y' is inaccessible due to its protection level.
```

```
'Point.x' is inaccessible due to its protection level.
```

C# language specification

For more information, see [Declared accessibility](#) in the [C# Language Specification](#). The language specification is the definitive source for C# syntax and usage.

See also

- [C# Reference](#)
- [C# Programming Guide](#)
- [C# Keywords](#)
- [Access Modifiers](#)
- [Accessibility Levels](#)
- [Modifiers](#)
- [public](#)
- [private](#)
- [internal](#)
- [Security concerns for internal virtual keywords](#)

public (C# Reference)

12/28/2021 • 2 minutes to read • [Edit Online](#)

The `public` keyword is an access modifier for types and type members. Public access is the most permissive access level. There are no restrictions on accessing public members, as in this example:

```
class SampleClass
{
    public int x; // No access restrictions.
}
```

See [Access Modifiers](#) and [Accessibility Levels](#) for more information.

Example

In the following example, two classes are declared, `PointTest` and `Program`. The public members `x` and `y` of `PointTest` are accessed directly from `Program`.

```
class PointTest
{
    public int x;
    public int y;
}

class Program
{
    static void Main()
    {
        var p = new PointTest();
        // Direct access to public members.
        p.x = 10;
        p.y = 15;
        Console.WriteLine($"x = {p.x}, y = {p.y}");
    }
}
// Output: x = 10, y = 15
```

If you change the `public` access level to `private` or `protected`, you will get the error message:

'PointTest.y' is inaccessible due to its protection level.

C# language specification

For more information, see [Declared accessibility](#) in the [C# Language Specification](#). The language specification is the definitive source for C# syntax and usage.

See also

- [C# Reference](#)
- [C# Programming Guide](#)
- [Access Modifiers](#)
- [C# Keywords](#)
- [Access Modifiers](#)

- Accessibility Levels
- Modifiers
- private
- protected
- internal

protected internal (C# Reference)

12/28/2021 • 2 minutes to read • [Edit Online](#)

The `protected internal` keyword combination is a member access modifier. A protected internal member is accessible from the current assembly or from types that are derived from the containing class. For a comparison of `protected internal` with the other access modifiers, see [Accessibility Levels](#).

Example

A protected internal member of a base class is accessible from any type within its containing assembly. It is also accessible in a derived class located in another assembly only if the access occurs through a variable of the derived class type. For example, consider the following code segment:

```
// Assembly1.cs
// Compile with: /target:library
public class BaseClass
{
    protected internal int myValue = 0;
}

class TestAccess
{
    void Access()
    {
        var baseObject = new BaseClass();
        baseObject.myValue = 5;
    }
}
```

```
// Assembly2.cs
// Compile with: /reference:Assembly1.dll
class DerivedClass : BaseClass
{
    static void Main()
    {
        var baseObject = new BaseClass();
        var derivedObject = new DerivedClass();

        // Error CS1540, because myValue can only be accessed by
        // classes derived from BaseClass.
        // baseObject.myValue = 10;

        // OK, because this class derives from BaseClass.
        derivedObject.myValue = 10;
    }
}
```

This example contains two files, `Assembly1.cs` and `Assembly2.cs`. The first file contains a public base class, `BaseClass`, and another class, `TestAccess`. `BaseClass` owns a protected internal member, `myValue`, which is accessed by the `TestAccess` type. In the second file, an attempt to access `myValue` through an instance of `BaseClass` will produce an error, while an access to this member through an instance of a derived class, `DerivedClass` will succeed.

Struct members cannot be `protected internal` because the struct cannot be inherited.

C# language specification

For more information, see the [C# Language Specification](#). The language specification is the definitive source for C# syntax and usage.

See also

- [C# Reference](#)
- [C# Programming Guide](#)
- [C# Keywords](#)
- [Access Modifiers](#)
- [Accessibility Levels](#)
- [Modifiers](#)
- [public](#)
- [private](#)
- [internal](#)
- [Security concerns for internal virtual keywords](#)

private protected (C# Reference)

12/28/2021 • 2 minutes to read • [Edit Online](#)

The `private protected` keyword combination is a member access modifier. A private protected member is accessible by types derived from the containing class, but only within its containing assembly. For a comparison of `private protected` with the other access modifiers, see [Accessibility Levels](#).

NOTE

The `private protected` access modifier is valid in C# version 7.2 and later.

Example

A private protected member of a base class is accessible from derived types in its containing assembly only if the static type of the variable is the derived class type. For example, consider the following code segment:

```
public class BaseClass
{
    private protected int myValue = 0;
}

public class DerivedClass1 : BaseClass
{
    void Access()
    {
        var baseObject = new BaseClass();

        // Error CS1540, because myValue can only be accessed by
        // classes derived from BaseClass.
        // baseObject.myValue = 5;

        // OK, accessed through the current derived class instance
        myValue = 5;
    }
}
```

```
// Assembly2.cs
// Compile with: /reference:Assembly1.dll
class DerivedClass2 : BaseClass
{
    void Access()
    {
        // Error CS0122, because myValue can only be
        // accessed by types in Assembly1
        // myValue = 10;
    }
}
```

This example contains two files, `Assembly1.cs` and `Assembly2.cs`. The first file contains a public base class, `BaseClass`, and a type derived from it, `DerivedClass1`. `BaseClass` owns a private protected member, `myValue`, which `DerivedClass1` tries to access in two ways. The first attempt to access `myValue` through an instance of `BaseClass` will produce an error. However, the attempt to use it as an inherited member in `DerivedClass1` will succeed.

In the second file, an attempt to access `myValue` as an inherited member of `DerivedClass2` will produce an error, as it is only accessible by derived types in `Assembly1`.

If `Assembly1.cs` contains an [InternalsVisibleToAttribute](#) that names `Assembly2`, the derived class `DerivedClass2` will have access to `private protected` members declared in `BaseClass`. `InternalsVisibleTo` makes `private protected` members visible to derived classes in other assemblies.

Struct members cannot be `private protected` because the struct cannot be inherited.

C# language specification

For more information, see the [C# Language Specification](#). The language specification is the definitive source for C# syntax and usage.

See also

- [C# Reference](#)
- [C# Programming Guide](#)
- [C# Keywords](#)
- [Access Modifiers](#)
- [Accessibility Levels](#)
- [Modifiers](#)
- [public](#)
- [private](#)
- [internal](#)
- [Security concerns for internal virtual keywords](#)

abstract (C# Reference)

12/28/2021 • 3 minutes to read • [Edit Online](#)

The `abstract` modifier indicates that the thing being modified has a missing or incomplete implementation. The `abstract` modifier can be used with classes, methods, properties, indexers, and events. Use the `abstract` modifier in a class declaration to indicate that a class is intended only to be a base class of other classes, not instantiated on its own. Members marked as `abstract` must be implemented by non-`abstract` classes that derive from the `abstract` class.

Example 1

In this example, the class `Square` must provide an implementation of `GetArea` because it derives from `Shape`:

```
abstract class Shape
{
    public abstract int GetArea();
}

class Square : Shape
{
    private int _side;

    public Square(int n) => _side = n;

    // GetArea method is required to avoid a compile-time error.
    public override int GetArea() => _side * _side;

    static void Main()
    {
        var sq = new Square(12);
        Console.WriteLine($"Area of the square = {sq.GetArea()}");
    }
}

// Output: Area of the square = 144
```

Abstract classes have the following features:

- An `abstract` class cannot be instantiated.
- An `abstract` class may contain `abstract` methods and accessors.
- It is not possible to modify an `abstract` class with the `sealed` modifier because the two modifiers have opposite meanings. The `sealed` modifier prevents a class from being inherited and the `abstract` modifier requires a class to be inherited.
- A non-`abstract` class derived from an `abstract` class must include actual implementations of all inherited `abstract` methods and accessors.

Use the `abstract` modifier in a method or property declaration to indicate that the method or property does not contain implementation.

`Abstract` methods have the following features:

- An `abstract` method is implicitly a virtual method.
- `Abstract` method declarations are only permitted in `abstract` classes.

- Because an abstract method declaration provides no actual implementation, there is no method body; the method declaration simply ends with a semicolon and there are no curly braces ({ }) following the signature. For example:

```
public abstract void MyMethod();
```

The implementation is provided by a method [override](#), which is a member of a non-abstract class.

- It is an error to use the [static](#) or [virtual](#) modifiers in an abstract method declaration.

Abstract properties behave like abstract methods, except for the differences in declaration and invocation syntax.

- It is an error to use the `abstract` modifier on a static property.
- An abstract inherited property can be overridden in a derived class by including a property declaration that uses the [override](#) modifier.

For more information about abstract classes, see [Abstract and Sealed Classes and Class Members](#).

An abstract class must provide implementation for all interface members.

An abstract class that implements an interface might map the interface methods onto abstract methods. For example:

```
interface I
{
    void M();
}

abstract class C : I
{
    public abstract void M();
}
```

Example 2

In this example, the class `DerivedClass` is derived from an abstract class `BaseClass`. The abstract class contains an abstract method, `AbstractMethod`, and two abstract properties, `x` and `y`.

```
// Abstract class
abstract class BaseClass
{
    protected int _x = 100;
    protected int _y = 150;

    // Abstract method
    public abstract void AbstractMethod();

    // Abstract properties
    public abstract int X { get; }
    public abstract int Y { get; }
}

class DerivedClass : BaseClass
{
    public override void AbstractMethod()
    {
        _x++;
        _y++;
    }

    public override int X    // overriding property
    {
        get
        {
            return _x + 10;
        }
    }

    public override int Y    // overriding property
    {
        get
        {
            return _y + 10;
        }
    }

    static void Main()
    {
        var o = new DerivedClass();
        o.AbstractMethod();
        Console.WriteLine($"x = {o.X}, y = {o.Y}");
    }
}
// Output: x = 111, y = 161
```

In the preceding example, if you attempt to instantiate the abstract class by using a statement like this:

```
BaseClass bc = new BaseClass();    // Error
```

You will get an error saying that the compiler cannot create an instance of the abstract class 'BaseClass'.

C# Language Specification

For more information, see the [C# Language Specification](#). The language specification is the definitive source for C# syntax and usage.

See also

- [C# Reference](#)
- [C# Programming Guide](#)

- [Modifiers](#)
- [virtual](#)
- [override](#)
- [C# Keywords](#)

async (C# Reference)

12/28/2021 • 4 minutes to read • [Edit Online](#)

Use the `async` modifier to specify that a method, [lambda expression](#), or [anonymous method](#) is asynchronous. If you use this modifier on a method or expression, it's referred to as an *async method*. The following example defines an async method named `ExampleMethodAsync`:

```
public async Task<int> ExampleMethodAsync()
{
    //...
}
```

If you're new to asynchronous programming or do not understand how an async method uses the `await` operator to do potentially long-running work without blocking the caller's thread, read the introduction in [Asynchronous programming with async and await](#). The following code is found inside an async method and calls the `HttpClient.GetStringAsync` method:

```
string contents = await httpClient.GetStringAsync(requestUrl);
```

An async method runs synchronously until it reaches its first `await` expression, at which point the method is suspended until the awaited task is complete. In the meantime, control returns to the caller of the method, as the example in the next section shows.

If the method that the `async` keyword modifies doesn't contain an `await` expression or statement, the method executes synchronously. A compiler warning alerts you to any async methods that don't contain `await` statements, because that situation might indicate an error. See [Compiler Warning \(level 1\) CS4014](#).

The `async` keyword is contextual in that it's a keyword only when it modifies a method, a lambda expression, or an anonymous method. In all other contexts, it's interpreted as an identifier.

Example

The following example shows the structure and flow of control between an async event handler, `StartButton_Click`, and an async method, `ExampleMethodAsync`. The result from the async method is the number of characters of a web page. The code is suitable for a Windows Presentation Foundation (WPF) app or Windows Store app that you create in Visual Studio; see the code comments for setting up the app.

You can run this code in Visual Studio as a Windows Presentation Foundation (WPF) app or a Windows Store app. You need a Button control named `StartButton` and a Textbox control named `ResultsTextBox`. Remember to set the names and handler so that you have something like this:

```
<Button Content="Button" HorizontalAlignment="Left" Margin="88,77,0,0" VerticalAlignment="Top" Width="75"
        Click="StartButton_Click" Name="StartButton"/>
<TextBox HorizontalAlignment="Left" Height="137" Margin="88,140,0,0" TextWrapping="Wrap"
        Text="&lt;Enter a URL&gt;" VerticalAlignment="Top" Width="310" Name="ResultsTextBox"/>
```

To run the code as a WPF app:

- Paste this code into the `MainWindow` class in `MainWindow.xaml.cs`.
- Add a reference to `System.Net.Http`.

- Add a `using` directive for `System.Net.Http`.

To run the code as a Windows Store app:

- Paste this code into the `MainPage` class in `MainPage.xaml.cs`.
- Add using directives for `System.Net.Http` and `System.Threading.Tasks`.

```
private async void StartButton_Click(object sender, RoutedEventArgs e)
{
    // ExampleMethodAsync returns a Task<int>, which means that the method
    // eventually produces an int result. However, ExampleMethodAsync returns
    // the Task<int> value as soon as it reaches an await.
    ResultsTextBox.Text += "\n";

    try
    {
        int length = await ExampleMethodAsync();
        // Note that you could put "await ExampleMethodAsync()" in the next line where
        // "length" is, but due to when '+= ' fetches the value of ResultsTextBox, you
        // would not see the global side effect of ExampleMethodAsync setting the text.
        ResultsTextBox.Text += String.Format("Length: {0:N0}\n", length);
    }
    catch (Exception)
    {
        // Process the exception if one occurs.
    }
}

public async Task<int> ExampleMethodAsync()
{
    var httpClient = new HttpClient();
    int exampleInt = (await httpClient.GetStringAsync("http://msdn.microsoft.com")).Length;
    ResultsTextBox.Text += "Preparing to finish ExampleMethodAsync.\n";
    // After the following return statement, any method that's awaiting
    // ExampleMethodAsync (in this case, StartButton_Click) can get the
    // integer result.
    return exampleInt;
}

// The example displays the following output:
// Preparing to finish ExampleMethodAsync.
// Length: 53292
```

IMPORTANT

For more information about tasks and the code that executes while waiting for a task, see [Asynchronous programming with async and await](#). For a full console example that uses similar elements, see [Process asynchronous tasks as they complete \(C#\)](#).

Return Types

An async method can have the following return types:

- [Task](#)
- [Task<TResult>](#)
- **void.** `async void` methods are generally discouraged for code other than event handlers because callers cannot `await` those methods and must implement a different mechanism to report successful completion or error conditions.
- Starting with C# 7.0, any type that has an accessible `GetAwaiter` method. The `System.Threading.Tasks.ValueTask<TResult>` type is one such implementation. It is available by adding the

NuGet package `System.Threading.Tasks.Extensions`.

The async method can't declare any [in](#), [ref](#) or [out](#) parameters, nor can it have a [reference return value](#), but it can call methods that have such parameters.

You specify `Task<TResult>` as the return type of an async method if the [return](#) statement of the method specifies an operand of type `TResult`. You use `Task` if no meaningful value is returned when the method is completed. That is, a call to the method returns a `Task`, but when the `Task` is completed, any `await` expression that's awaiting the `Task` evaluates to `void`.

You use the `void` return type primarily to define event handlers, which require that return type. The caller of a `void`-returning async method can't await it and can't catch exceptions that the method throws.

Starting with C# 7.0, you return another type, typically a value type, that has a `GetAwaiter` method to minimize memory allocations in performance-critical sections of code.

For more information and examples, see [Async Return Types](#).

See also

- [AsyncStateMachineAttribute](#)
- [await](#)
- [Asynchronous programming with async and await](#)
- [Process asynchronous tasks as they complete](#)

const (C# Reference)

12/28/2021 • 2 minutes to read • [Edit Online](#)

You use the `const` keyword to declare a constant field or a constant local. Constant fields and locals aren't variables and may not be modified. Constants can be numbers, Boolean values, strings, or a null reference. Don't create a constant to represent information that you expect to change at any time. For example, don't use a constant field to store the price of a service, a product version number, or the brand name of a company. These values can change over time, and because compilers propagate constants, other code compiled with your libraries will have to be recompiled to see the changes. See also the [readonly](#) keyword. For example:

```
const int X = 0;
public const double GravitationalConstant = 6.673e-11;
private const string ProductName = "Visual C#";
```

Beginning with C# 10, [interpolated strings](#) may be constants, if all expressions used are also constant strings. This feature can improve the code that builds constant strings:

```
const string Language = "C#";
const string Platform = ".NET";
const string Version = "10.0";
const string FullProductName = $"{Platform} - Language: {Language} Version: {Version}";
```

Remarks

The type of a constant declaration specifies the type of the members that the declaration introduces. The initializer of a constant local or a constant field must be a constant expression that can be implicitly converted to the target type.

A constant expression is an expression that can be fully evaluated at compile time. Therefore, the only possible values for constants of reference types are `string` and a null reference.

The constant declaration can declare multiple constants, such as:

```
public const double X = 1.0, Y = 2.0, Z = 3.0;
```

The `static` modifier is not allowed in a constant declaration.

A constant can participate in a constant expression, as follows:

```
public const int C1 = 5;
public const int C2 = C1 + 100;
```

NOTE

The `readonly` keyword differs from the `const` keyword. A `const` field can only be initialized at the declaration of the field. A `readonly` field can be initialized either at the declaration or in a constructor. Therefore, `readonly` fields can have different values depending on the constructor used. Also, although a `const` field is a compile-time constant, the `readonly` field can be used for run-time constants, as in this line:

```
public static readonly uint l1 = (uint)DateTime.Now.Ticks;
```

Examples

```
public class ConstTest
{
    class SampleClass
    {
        public int x;
        public int y;
        public const int C1 = 5;
        public const int C2 = C1 + 5;

        public SampleClass(int p1, int p2)
        {
            x = p1;
            y = p2;
        }
    }

    static void Main()
    {
        var mC = new SampleClass(11, 22);
        Console.WriteLine($"x = {mC.x}, y = {mC.y}");
        Console.WriteLine($"C1 = {SampleClass.C1}, C2 = {SampleClass.C2}");
    }
}

/* Output
x = 11, y = 22
C1 = 5, C2 = 10
*/
```

This example demonstrates how to use constants as local variables.

```
public class SealedTest
{
    static void Main()
    {
        const int C = 707;
        Console.WriteLine($"My local constant = {C}");
    }
}

// Output: My local constant = 707
```

C# language specification

For more information, see the [C# Language Specification](#). The language specification is the definitive source for C# syntax and usage.

See also

- [C# Reference](#)

- [C# Programming Guide](#)
- [C# Keywords](#)
- [Modifiers](#)
- [readonly](#)

event (C# reference)

12/28/2021 • 2 minutes to read • [Edit Online](#)

The `event` keyword is used to declare an event in a publisher class.

Example

The following example shows how to declare and raise an event that uses [EventHandler](#) as the underlying delegate type. For the complete code example that also shows how to use the generic [EventHandler<TEventArgs>](#) delegate type and how to subscribe to an event and create an event handler method, see [How to publish events that conform to .NET Guidelines](#).

```
public class SampleEventArgs
{
    public SampleEventArgs(string text) { Text = text; }
    public string Text { get; } // readonly
}

public class Publisher
{
    // Declare the delegate (if using non-generic pattern).
    public delegate void SampleEventHandler(object sender, SampleEventArgs e);

    // Declare the event.
    public event SampleEventHandler SampleEvent;

    // Wrap the event in a protected virtual method
    // to enable derived classes to raise the event.
    protected virtual void RaiseSampleEvent()
    {
        // Raise the event in a thread-safe manner using the ?. operator.
        SampleEvent?.Invoke(this, new SampleEventArgs("Hello"));
    }
}
```

Events are a special kind of multicast delegate that can only be invoked from within the class or struct where they are declared (the publisher class). If other classes or structs subscribe to the event, their event handler methods will be called when the publisher class raises the event. For more information and code examples, see [Events](#) and [Delegates](#).

Events can be marked as [public](#), [private](#), [protected](#), [internal](#), [protected internal](#), or [private protected](#). These access modifiers define how users of the class can access the event. For more information, see [Access Modifiers](#).

Keywords and events

The following keywords apply to events.

KEYWORD	DESCRIPTION	FOR MORE INFORMATION
static	Makes the event available to callers at any time, even if no instance of the class exists.	Static Classes and Static Class Members

KEYWORD	DESCRIPTION	FOR MORE INFORMATION
virtual	Allows derived classes to override the event behavior by using the override keyword.	Inheritance
sealed	Specifies that for derived classes it is no longer virtual.	
abstract	The compiler will not generate the <code>add</code> and <code>remove</code> event accessor blocks and therefore derived classes must provide their own implementation.	

An event may be declared as a static event by using the [static](#) keyword. This makes the event available to callers at any time, even if no instance of the class exists. For more information, see [Static Classes and Static Class Members](#).

An event can be marked as a virtual event by using the [virtual](#) keyword. This enables derived classes to override the event behavior by using the [override](#) keyword. For more information, see [Inheritance](#). An event overriding a virtual event can also be [sealed](#), which specifies that for derived classes it is no longer virtual. Lastly, an event can be declared [abstract](#), which means that the compiler will not generate the `add` and `remove` event accessor blocks. Therefore derived classes must provide their own implementation.

C# language specification

For more information, see the [C# Language Specification](#). The language specification is the definitive source for C# syntax and usage.

See also

- [C# Reference](#)
- [C# Programming Guide](#)
- [C# Keywords](#)
- [add](#)
- [remove](#)
- [Modifiers](#)
- [How to combine delegates \(Multicast Delegates\)](#)

extern (C# Reference)

12/28/2021 • 2 minutes to read • [Edit Online](#)

The `extern` modifier is used to declare a method that is implemented externally. A common use of the `extern` modifier is with the `DllImport` attribute when you are using Interop services to call into unmanaged code. In this case, the method must also be declared as `static`, as shown in the following example:

```
[DllImport("avifil32.dll")]
private static extern void AVIFileInit();
```

The `extern` keyword can also define an external assembly alias, which makes it possible to reference different versions of the same component from within a single assembly. For more information, see [extern alias](#).

It is an error to use the `abstract` and `extern` modifiers together to modify the same member. Using the `extern` modifier means that the method is implemented outside the C# code, whereas using the `abstract` modifier means that the method implementation is not provided in the class.

The `extern` keyword has more limited uses in C# than in C++. To compare the C# keyword with the C++ keyword, see [Using extern to Specify Linkage in the C++ Language Reference](#).

Example 1

In this example, the program receives a string from the user and displays it inside a message box. The program uses the `MessageBox` method imported from the `User32.dll` library.

```
//using System.Runtime.InteropServices;
class ExternTest
{
    [DllImport("User32.dll", CharSet=CharSet.Unicode)]
    public static extern int MessageBox(IntPtr h, string m, string c, int type);

    static int Main()
    {
        string myString;
        Console.WriteLine("Enter your message: ");
        myString = Console.ReadLine();
        return MessageBox((IntPtr)0, myString, "My Message Box", 0);
    }
}
```

Example 2

This example illustrates a C# program that calls into a C library (a native DLL).

1. Create the following C file and name it `cmdll.c`:

```
// cmdll.c
// Compile with: -LD
int __declspec(dllexport) SampleMethod(int i)
{
    return i*10;
}
```

2. Open a Visual Studio x64 (or x32) Native Tools Command Prompt window from the Visual Studio installation directory and compile the `cmdll.c` file by typing `cl -LD cmdll.c` at the command prompt.
3. In the same directory, create the following C# file and name it `cm.cs` :

```
// cm.cs
using System;
using System.Runtime.InteropServices;
public class MainClass
{
    [DllImport("Cmdll.dll")]
    public static extern int SampleMethod(int x);

    static void Main()
    {
        Console.WriteLine("SampleMethod() returns {0}.", SampleMethod(5));
    }
}
```

4. Open a Visual Studio x64 (or x32) Native Tools Command Prompt window from the Visual Studio installation directory and compile the `cm.cs` file by typing:

```
csc cm.cs (for the x64 command prompt) —or— csc -platform:x86 cm.cs (for the x32 command prompt)
```

This will create the executable file `cm.exe` .

5. Run `cm.exe` . The `SampleMethod` method passes the value 5 to the DLL file, which returns the value multiplied by 10. The program produces the following output:

```
SampleMethod() returns 50.
```

C# language specification

For more information, see the [C# Language Specification](#). The language specification is the definitive source for C# syntax and usage.

See also

- [System.Runtime.InteropServices.DllImportAttribute](#)
- [C# Reference](#)
- [C# Programming Guide](#)
- [C# Keywords](#)
- [Modifiers](#)

in (Generic Modifier) (C# Reference)

12/28/2021 • 2 minutes to read • [Edit Online](#)

For generic type parameters, the `in` keyword specifies that the type parameter is contravariant. You can use the `in` keyword in generic interfaces and delegates.

Contravariance enables you to use a less derived type than that specified by the generic parameter. This allows for implicit conversion of classes that implement contravariant interfaces and implicit conversion of delegate types. Covariance and contravariance in generic type parameters are supported for reference types, but they are not supported for value types.

A type can be declared contravariant in a generic interface or delegate only if it defines the type of a method's parameters and not of a method's return type. `In`, `ref`, and `out` parameters must be invariant, meaning they are neither covariant or contravariant.

An interface that has a contravariant type parameter allows its methods to accept arguments of less derived types than those specified by the interface type parameter. For example, in the `IComparer<T>` interface, type `T` is contravariant, you can assign an object of the `IComparer<Person>` type to an object of the `IComparer<Employee>` type without using any special conversion methods if `Employee` inherits `Person`.

A contravariant delegate can be assigned another delegate of the same type, but with a less derived generic type parameter.

For more information, see [Covariance and Contravariance](#).

Contravariant generic interface

The following example shows how to declare, extend, and implement a contravariant generic interface. It also shows how you can use implicit conversion for classes that implement this interface.

```
// Contravariant interface.
interface IContravariant<in A> { }

// Extending contravariant interface.
interface IExtContravariant<in A> : IContravariant<A> { }

// Implementing contravariant interface.
class Sample<A> : IContravariant<A> { }

class Program
{
    static void Test()
    {
        IContravariant<Object> iobj = new Sample<Object>();
        IContravariant<String> istr = new Sample<String>();

        // You can assign iobj to istr because
        // the IContravariant interface is contravariant.
        istr = iobj;
    }
}
```

Contravariant generic delegate

The following example shows how to declare, instantiate, and invoke a contravariant generic delegate. It also

shows how you can implicitly convert a delegate type.

```
// Contravariant delegate.
public delegate void DContravariant<in A>(A argument);

// Methods that match the delegate signature.
public static void SampleControl(Control control)
{ }
public static void SampleButton(Button button)
{ }

public void Test()
{

    // Instantiating the delegates with the methods.
    DContravariant<Control> dControl = SampleControl;
    DContravariant<Button> dButton = SampleButton;

    // You can assign dControl to dButton
    // because the DContravariant delegate is contravariant.
    dButton = dControl;

    // Invoke the delegate.
    dButton(new Button());
}
```

C# language specification

For more information, see the [C# Language Specification](#). The language specification is the definitive source for C# syntax and usage.

See also

- [out](#)
- [Covariance and Contravariance](#)
- [Modifiers](#)

new modifier (C# Reference)

12/28/2021 • 3 minutes to read • [Edit Online](#)

When used as a declaration modifier, the `new` keyword explicitly hides a member that is inherited from a base class. When you hide an inherited member, the derived version of the member replaces the base class version. This assumes that the base class version of the member is visible, as it would already be hidden if it were marked as `private` or, in some cases, `internal`. Although you can hide `public` or `protected` members without using the `new` modifier, you get a compiler warning. If you use `new` to explicitly hide a member, it suppresses this warning.

You can also use the `new` keyword to [create an instance of a type](#) or as a [generic type constraint](#).

To hide an inherited member, declare it in the derived class by using the same member name, and modify it with the `new` keyword. For example:

```
public class BaseC
{
    public int x;
    public void Invoke() { }
}
public class DerivedC : BaseC
{
    new public void Invoke() { }
}
```

In this example, `BaseC.Invoke` is hidden by `DerivedC.Invoke`. The field `x` is not affected because it is not hidden by a similar name.

Name hiding through inheritance takes one of the following forms:

- Generally, a constant, field, property, or type that is introduced in a class or struct hides all base class members that share its name. There are special cases. For example, if you declare a new field with name `N` to have a type that is not invocable, and a base type declares `N` to be a method, the new field does not hide the base declaration in invocation syntax. For more information, see the [Member lookup](#) section of the [C# language specification](#).
- A method introduced in a class or struct hides properties, fields, and types that share that name in the base class. It also hides all base class methods that have the same signature.
- An indexer introduced in a class or struct hides all base class indexers that have the same signature.

It is an error to use both `new` and `override` on the same member, because the two modifiers have mutually exclusive meanings. The `new` modifier creates a new member with the same name and causes the original member to become hidden. The `override` modifier extends the implementation for an inherited member.

Using the `new` modifier in a declaration that does not hide an inherited member generates a warning.

Examples

In this example, a base class, `BaseC`, and a derived class, `DerivedC`, use the same field name `x`, which hides the value of the inherited field. The example demonstrates the use of the `new` modifier. It also demonstrates how to access the hidden members of the base class by using their fully qualified names.

```

public class BaseC
{
    public static int x = 55;
    public static int y = 22;
}

public class DerivedC : BaseC
{
    // Hide field 'x'.
    new public static int x = 100;

    static void Main()
    {
        // Display the new value of x:
        Console.WriteLine(x);

        // Display the hidden value of x:
        Console.WriteLine(BaseC.x);

        // Display the unhidden member y:
        Console.WriteLine(y);
    }
}
/*
Output:
100
55
22
*/

```

In this example, a nested class hides a class that has the same name in the base class. The example demonstrates how to use the `new` modifier to eliminate the warning message and how to access the hidden class members by using their fully qualified names.

```

public class BaseC
{
    public class NestedC
    {
        public int x = 200;
        public int y;
    }
}

public class DerivedC : BaseC
{
    // Nested type hiding the base type members.
    new public class NestedC
    {
        public int x = 100;
        public int y;
        public int z;
    }

    static void Main()
    {
        // Creating an object from the overlapping class:
        NestedC c1 = new NestedC();

        // Creating an object from the hidden class:
        BaseC.NestedC c2 = new BaseC.NestedC();

        Console.WriteLine(c1.x);
        Console.WriteLine(c2.x);
    }
}
/*
Output:
100
200
*/

```

If you remove the `new` modifier, the program will still compile and run, but you will get the following warning:

The keyword `new` is required on 'MyDerivedC.x' because it hides inherited member 'MyBaseC.x'.

C# language specification

For more information, see [The new modifier](#) section of the [C# language specification](#).

See also

- [C# Reference](#)
- [C# Programming Guide](#)
- [C# Keywords](#)
- [Modifiers](#)
- [Versioning with the Override and New Keywords](#)
- [Knowing When to Use Override and New Keywords](#)

out (generic modifier) (C# Reference)

12/28/2021 • 2 minutes to read • [Edit Online](#)

For generic type parameters, the `out` keyword specifies that the type parameter is covariant. You can use the `out` keyword in generic interfaces and delegates.

Covariance enables you to use a more derived type than that specified by the generic parameter. This allows for implicit conversion of classes that implement covariant interfaces and implicit conversion of delegate types. Covariance and contravariance are supported for reference types, but they are not supported for value types.

An interface that has a covariant type parameter enables its methods to return more derived types than those specified by the type parameter. For example, because in .NET Framework 4, in `IEnumerable<T>`, type `T` is covariant, you can assign an object of the `IEnumerable(Of String)` type to an object of the `IEnumerable(Of Object)` type without using any special conversion methods.

A covariant delegate can be assigned another delegate of the same type, but with a more derived generic type parameter.

For more information, see [Covariance and Contravariance](#).

Example - covariant generic interface

The following example shows how to declare, extend, and implement a covariant generic interface. It also shows how to use implicit conversion for classes that implement a covariant interface.

```
// Covariant interface.
interface ICovariant<out R> { }

// Extending covariant interface.
interface IExtCovariant<out R> : ICovariant<R> { }

// Implementing covariant interface.
class Sample<R> : ICovariant<R> { }

class Program
{
    static void Test()
    {
        ICovariant<Object> iobj = new Sample<Object>();
        ICovariant<String> istr = new Sample<String>();

        // You can assign istr to iobj because
        // the ICovariant interface is covariant.
        iobj = istr;
    }
}
```

In a generic interface, a type parameter can be declared covariant if it satisfies the following conditions:

- The type parameter is used only as a return type of interface methods and not used as a type of method arguments.

NOTE

There is one exception to this rule. If in a covariant interface you have a contravariant generic delegate as a method parameter, you can use the covariant type as a generic type parameter for this delegate. For more information about covariant and contravariant generic delegates, see [Variance in Delegates](#) and [Using Variance for Func and Action Generic Delegates](#).

- The type parameter is not used as a generic constraint for the interface methods.

Example - covariant generic delegate

The following example shows how to declare, instantiate, and invoke a covariant generic delegate. It also shows how to implicitly convert delegate types.

```
// Covariant delegate.
public delegate R DCovariant<out R>();

// Methods that match the delegate signature.
public static Control SampleControl()
{ return new Control(); }

public static Button SampleButton()
{ return new Button(); }

public void Test()
{
    // Instantiate the delegates with the methods.
    DCovariant<Control> dControl = SampleControl;
    DCovariant<Button> dButton = SampleButton;

    // You can assign dButton to dControl
    // because the DCovariant delegate is covariant.
    dControl = dButton;

    // Invoke the delegate.
    dControl();
}
```

In a generic delegate, a type can be declared covariant if it is used only as a method return type and not used for method arguments.

C# language specification

For more information, see the [C# Language Specification](#). The language specification is the definitive source for C# syntax and usage.

See also

- [Variance in Generic Interfaces](#)
- [in](#)
- [Modifiers](#)

override (C# reference)

12/28/2021 • 3 minutes to read • [Edit Online](#)

The `override` modifier is required to extend or modify the abstract or virtual implementation of an inherited method, property, indexer, or event.

In the following example, the `Square` class must provide an overridden implementation of `GetArea` because `GetArea` is inherited from the abstract `Shape` class:

```
abstract class Shape
{
    public abstract int GetArea();
}

class Square : Shape
{
    private int _side;

    public Square(int n) => _side = n;

    // GetArea method is required to avoid a compile-time error.
    public override int GetArea() => _side * _side;

    static void Main()
    {
        var sq = new Square(12);
        Console.WriteLine($"Area of the square = {sq.GetArea()}");
    }
}
// Output: Area of the square = 144
```

An `override` method provides a new implementation of the method inherited from a base class. The method that is overridden by an `override` declaration is known as the overridden base method. An `override` method must have the same signature as the overridden base method. Beginning with C# 9.0, `override` methods support covariant return types. In particular, the return type of an `override` method can derive from the return type of the corresponding base method. In C# 8.0 and earlier, the return types of an `override` method and the overridden base method must be the same.

You cannot override a non-virtual or static method. The overridden base method must be `virtual`, `abstract`, or `override`.

An `override` declaration cannot change the accessibility of the `virtual` method. Both the `override` method and the `virtual` method must have the same [access level modifier](#).

You cannot use the `new`, `static`, or `virtual` modifiers to modify an `override` method.

An overriding property declaration must specify exactly the same access modifier, type, and name as the inherited property. Beginning with C# 9.0, read-only overriding properties support covariant return types. The overridden property must be `virtual`, `abstract`, or `override`.

For more information about how to use the `override` keyword, see [Versioning with the Override and New Keywords](#) and [Knowing when to use Override and New Keywords](#). For information about inheritance, see [Inheritance](#).

Example

This example defines a base class named `Employee`, and a derived class named `SalesEmployee`. The `SalesEmployee` class includes an extra field, `salesbonus`, and overrides the method `CalculatePay` in order to take it into account.

```

class TestOverride
{
    public class Employee
    {
        public string Name { get; }

        // Basepay is defined as protected, so that it may be
        // accessed only by this class and derived classes.
        protected decimal _basepay;

        // Constructor to set the name and basepay values.
        public Employee(string name, decimal basepay)
        {
            Name = name;
            _basepay = basepay;
        }

        // Declared virtual so it can be overridden.
        public virtual decimal CalculatePay()
        {
            return _basepay;
        }
    }

    // Derive a new class from Employee.
    public class SalesEmployee : Employee
    {
        // New field that will affect the base pay.
        private decimal _salesbonus;

        // The constructor calls the base-class version, and
        // initializes the salesbonus field.
        public SalesEmployee(string name, decimal basepay, decimal salesbonus)
            : base(name, basepay)
        {
            _salesbonus = salesbonus;
        }

        // Override the CalculatePay method
        // to take bonus into account.
        public override decimal CalculatePay()
        {
            return _basepay + _salesbonus;
        }
    }

    static void Main()
    {
        // Create some new employees.
        var employee1 = new SalesEmployee("Alice", 1000, 500);
        var employee2 = new Employee("Bob", 1200);

        Console.WriteLine($"Employee1 {employee1.Name} earned: {employee1.CalculatePay()}");
        Console.WriteLine($"Employee2 {employee2.Name} earned: {employee2.CalculatePay()}");
    }
}
/*
Output:
Employee1 Alice earned: 1500
Employee2 Bob earned: 1200
*/

```

C# language specification

For more information, see the [Override methods](#) section of the [C# language specification](#).

For more information about covariant return types, see the [feature proposal note](#).

See also

- [C# reference](#)
- [Inheritance](#)
- [C# keywords](#)
- [Modifiers](#)
- [abstract](#)
- [virtual](#)
- [new \(modifier\)](#)
- [Polymorphism](#)

readonly (C# Reference)

12/28/2021 • 3 minutes to read • [Edit Online](#)

The `readonly` keyword is a modifier that can be used in four contexts:

- In a [field declaration](#), `readonly` indicates that assignment to the field can only occur as part of the declaration or in a constructor in the same class. A readonly field can be assigned and reassigned multiple times within the field declaration and constructor.

A `readonly` field can't be assigned after the constructor exits. This rule has different implications for value types and reference types:

- Because value types directly contain their data, a field that is a `readonly` value type is immutable.
- Because reference types contain a reference to their data, a field that is a `readonly` reference type must always refer to the same object. That object isn't immutable. The `readonly` modifier prevents the field from being replaced by a different instance of the reference type. However, the modifier doesn't prevent the instance data of the field from being modified through the read-only field.

WARNING

An externally visible type that contains an externally visible read-only field that is a mutable reference type may be a security vulnerability and may trigger warning CA2104 : "Do not declare read only mutable reference types."

- In a `readonly struct` type definition, `readonly` indicates that the structure type is immutable. For more information, see the [readonly struct](#) section of the [Structure types](#) article.
- In an instance member declaration within a structure type, `readonly` indicates that an instance member doesn't modify the state of the structure. For more information, see the [readonly instance members](#) section of the [Structure types](#) article.
- In a [ref readonly method return](#), the `readonly` modifier indicates that method returns a reference and writes aren't allowed to that reference.

The `readonly struct` and `ref readonly` contexts were added in C# 7.2. `readonly` struct members were added in C# 8.0

Readonly field example

In this example, the value of the field `year` can't be changed in the method `ChangeYear`, even though it's assigned a value in the class constructor:

```
class Age
{
    private readonly int _year;
    Age(int year)
    {
        _year = year;
    }
    void ChangeYear()
    {
        //_year = 1967; // Compile error if uncommented.
    }
}
```

You can assign a value to a `readonly` field only in the following contexts:

- When the variable is initialized in the declaration, for example:

```
public readonly int y = 5;
```

- In an instance constructor of the class that contains the instance field declaration.
- In the static constructor of the class that contains the static field declaration.

These constructor contexts are also the only contexts in which it's valid to pass a `readonly` field as an [out](#) or [ref](#) parameter.

NOTE

The `readonly` keyword is different from the `const` keyword. A `const` field can only be initialized at the declaration of the field. A `readonly` field can be assigned multiple times in the field declaration and in any constructor. Therefore, `readonly` fields can have different values depending on the constructor used. Also, while a `const` field is a compile-time constant, the `readonly` field can be used for run-time constants as in the following example:

```
public static readonly uint timeStamp = (uint)DateTime.Now.Ticks;
```

```

public class SamplePoint
{
    public int x;
    // Initialize a readonly field
    public readonly int y = 25;
    public readonly int z;

    public SamplePoint()
    {
        // Initialize a readonly instance field
        z = 24;
    }

    public SamplePoint(int p1, int p2, int p3)
    {
        x = p1;
        y = p2;
        z = p3;
    }

    public static void Main()
    {
        SamplePoint p1 = new SamplePoint(11, 21, 32);    // OK
        Console.WriteLine($"p1: x={p1.x}, y={p1.y}, z={p1.z}");
        SamplePoint p2 = new SamplePoint();
        p2.x = 55;    // OK
        Console.WriteLine($"p2: x={p2.x}, y={p2.y}, z={p2.z}");
    }
    /*
    Output:
        p1: x=11, y=21, z=32
        p2: x=55, y=25, z=24
    */
}

```

In the preceding example, if you use a statement like the following example:

```
p2.y = 66;    // Error
```

you'll get the compiler error message:

A readonly field cannot be assigned to (except in a constructor or a variable initializer)

Ref readonly return example

The `readonly` modifier on a `ref return` indicates that the returned reference can't be modified. The following example returns a reference to the origin. It uses the `readonly` modifier to indicate that callers can't modify the origin:

```

private static readonly SamplePoint s_origin = new SamplePoint(0, 0, 0);
public static ref readonly SamplePoint Origin => ref s_origin;

```

The type returned doesn't need to be a `readonly struct`. Any type that can be returned by `ref` can be returned by `ref readonly`.

C# language specification

For more information, see the [C# Language Specification](#). The language specification is the definitive source for C# syntax and usage.

You can also see the language specification proposals:

- [readonly ref and readonly struct](#)
- [readonly struct members](#)

See also

- [C# Reference](#)
- [C# Programming Guide](#)
- [C# Keywords](#)
- [Modifiers](#)
- [const](#)
- [Fields](#)

sealed (C# Reference)

12/28/2021 • 2 minutes to read • [Edit Online](#)

When applied to a class, the `sealed` modifier prevents other classes from inheriting from it. In the following example, class `B` inherits from class `A`, but no class can inherit from class `B`.

```
class A {}
sealed class B : A {}
```

You can also use the `sealed` modifier on a method or property that overrides a virtual method or property in a base class. This enables you to allow classes to derive from your class and prevent them from overriding specific virtual methods or properties.

Example

In the following example, `Z` inherits from `Y` but `Z` cannot override the virtual function `F` that is declared in `X` and sealed in `Y`.

```
class X
{
    protected virtual void F() { Console.WriteLine("X.F"); }
    protected virtual void F2() { Console.WriteLine("X.F2"); }
}

class Y : X
{
    sealed protected override void F() { Console.WriteLine("Y.F"); }
    protected override void F2() { Console.WriteLine("Y.F2"); }
}

class Z : Y
{
    // Attempting to override F causes compiler error CS0239.
    // protected override void F() { Console.WriteLine("Z.F"); }

    // Overriding F2 is allowed.
    protected override void F2() { Console.WriteLine("Z.F2"); }
}
```

When you define new methods or properties in a class, you can prevent deriving classes from overriding them by not declaring them as `virtual`.

It is an error to use the `abstract` modifier with a sealed class, because an abstract class must be inherited by a class that provides an implementation of the abstract methods or properties.

When applied to a method or property, the `sealed` modifier must always be used with `override`.

Because structs are implicitly sealed, they cannot be inherited.

For more information, see [Inheritance](#).

For more examples, see [Abstract and Sealed Classes and Class Members](#).

```
sealed class SealedClass
{
    public int x;
    public int y;
}

class SealedTest2
{
    static void Main()
    {
        var sc = new SealedClass();
        sc.x = 110;
        sc.y = 150;
        Console.WriteLine($"x = {sc.x}, y = {sc.y}");
    }
}
// Output: x = 110, y = 150
```

In the previous example, you might try to inherit from the sealed class by using the following statement:

```
class MyDerivedC: SealedClass {} // Error
```

The result is an error message:

```
'MyDerivedC': cannot derive from sealed type 'SealedClass'
```

Remarks

To determine whether to seal a class, method, or property, you should generally consider the following two points:

- The potential benefits that deriving classes might gain through the ability to customize your class.
- The potential that deriving classes could modify your classes in such a way that they would no longer work correctly or as expected.

C# language specification

For more information, see the [C# Language Specification](#). The language specification is the definitive source for C# syntax and usage.

See also

- [C# Reference](#)
- [C# Programming Guide](#)
- [C# Keywords](#)
- [Static Classes and Static Class Members](#)
- [Abstract and Sealed Classes and Class Members](#)
- [Access Modifiers](#)
- [Modifiers](#)
- [override](#)
- [virtual](#)

static (C# Reference)

12/28/2021 • 3 minutes to read • [Edit Online](#)

This page covers the `static` modifier keyword. The `static` keyword is also part of the `using static` directive.

Use the `static` modifier to declare a static member, which belongs to the type itself rather than to a specific object. The `static` modifier can be used to declare `static` classes. In classes, interfaces, and structs, you may add the `static` modifier to fields, methods, properties, operators, events, and constructors. The `static` modifier can't be used with indexers or finalizers. For more information, see [Static Classes and Static Class Members](#).

Beginning with C# 8.0, you can add the `static` modifier to a [local function](#). A static local function can't capture local variables or instance state.

Beginning with C# 9.0, you can add the `static` modifier to a [lambda expression](#) or [anonymous method](#). A static lambda or anonymous method can't capture local variables or instance state.

Example - static class

The following class is declared as `static` and contains only `static` methods:

```
static class CompanyEmployee
{
    public static void DoSomething() { /*...*/ }
    public static void DoSomethingElse() { /*...*/ }
}
```

A constant or type declaration is implicitly a `static` member. A `static` member can't be referenced through an instance. Instead, it's referenced through the type name. For example, consider the following class:

```
public class MyBaseC
{
    public struct MyStruct
    {
        public static int x = 100;
    }
}
```

To refer to the `static` member `x`, use the fully qualified name, `MyBaseC.MyStruct.x`, unless the member is accessible from the same scope:

```
Console.WriteLine(MyBaseC.MyStruct.x);
```

While an instance of a class contains a separate copy of all instance fields of the class, there's only one copy of each `static` field.

It isn't possible to use `this` to reference `static` methods or property accessors.

If the `static` keyword is applied to a class, all the members of the class must be `static`.

Classes, interfaces, and `static` classes may have `static` constructors. A `static` constructor is called at some point between when the program starts and the class is instantiated.

NOTE

The `static` keyword has more limited uses than in C++. To compare with the C++ keyword, see [Storage classes \(C++\)](#).

To demonstrate `static` members, consider a class that represents a company employee. Assume that the class contains a method to count employees and a field to store the number of employees. Both the method and the field don't belong to any one employee instance. Instead, they belong to the class of employees as a whole. They should be declared as `static` members of the class.

Example - static field and method

This example reads the name and ID of a new employee, increments the employee counter by one, and displays the information for the new employee and the new number of employees. This program reads the current number of employees from the keyboard.

```

public class Employee4
{
    public string id;
    public string name;

    public Employee4()
    {
    }

    public Employee4(string name, string id)
    {
        this.name = name;
        this.id = id;
    }

    public static int employeeCounter;

    public static int AddEmployee()
    {
        return ++employeeCounter;
    }
}

class MainClass : Employee4
{
    static void Main()
    {
        Console.Write("Enter the employee's name: ");
        string name = Console.ReadLine();
        Console.Write("Enter the employee's ID: ");
        string id = Console.ReadLine();

        // Create and configure the employee object.
        Employee4 e = new Employee4(name, id);
        Console.Write("Enter the current number of employees: ");
        string n = Console.ReadLine();
        Employee4.employeeCounter = Int32.Parse(n);
        Employee4.AddEmployee();

        // Display the new information.
        Console.WriteLine($"Name: {e.name}");
        Console.WriteLine($"ID: {e.id}");
        Console.WriteLine($"New Number of Employees: {Employee4.employeeCounter}");
    }
}
/*
Input:
Matthias Berndt
AF643G
15
*
Sample Output:
Enter the employee's name: Matthias Berndt
Enter the employee's ID: AF643G
Enter the current number of employees: 15
Name: Matthias Berndt
ID: AF643G
New Number of Employees: 16
*/

```

Example - static initialization

This example shows that you can initialize a `static` field by using another `static` field that is not yet declared. The results will be undefined until you explicitly assign a value to the `static` field.

```
class Test
{
    static int x = y;
    static int y = 5;

    static void Main()
    {
        Console.WriteLine(Test.x);
        Console.WriteLine(Test.y);

        Test.x = 99;
        Console.WriteLine(Test.x);
    }
}
/*
Output:
    0
    5
    99
*/
```

C# language specification

For more information, see the [C# Language Specification](#). The language specification is the definitive source for C# syntax and usage.

See also

- [C# Reference](#)
- [C# Programming Guide](#)
- [C# Keywords](#)
- [Modifiers](#)
- [using static directive](#)
- [Static Classes and Static Class Members](#)

unsafe (C# Reference)

12/28/2021 • 2 minutes to read • [Edit Online](#)

The `unsafe` keyword denotes an unsafe context, which is required for any operation involving pointers. For more information, see [Unsafe Code and Pointers](#).

You can use the `unsafe` modifier in the declaration of a type or a member. The entire textual extent of the type or member is therefore considered an unsafe context. For example, the following is a method declared with the `unsafe` modifier:

```
unsafe static void FastCopy(byte[] src, byte[] dst, int count)
{
    // Unsafe context: can use pointers here.
}
```

The scope of the unsafe context extends from the parameter list to the end of the method, so pointers can also be used in the parameter list:

```
unsafe static void FastCopy ( byte* ps, byte* pd, int count ) {...}
```

You can also use an unsafe block to enable the use of an unsafe code inside this block. For example:

```
unsafe
{
    // Unsafe context: can use pointers here.
}
```

To compile unsafe code, you must specify the [AllowUnsafeBlocks](#) compiler option. Unsafe code is not verifiable by the common language runtime.

Example

```
// compile with: -unsafe
class UnsafeTest
{
    // Unsafe method: takes pointer to int.
    unsafe static void SquarePtrParam(int* p)
    {
        *p *= *p;
    }

    unsafe static void Main()
    {
        int i = 5;
        // Unsafe method: uses address-of operator (&).
        SquarePtrParam(&i);
        Console.WriteLine(i);
    }
}
// Output: 25
```


C# language specification

For more information, see [Unsafe code](#) in the [C# Language Specification](#). The language specification is the definitive source for C# syntax and usage.

See also

- [C# Reference](#)
- [C# Programming Guide](#)
- [C# Keywords](#)
- [fixed Statement](#)
- [Unsafe Code and Pointers](#)
- [Fixed Size Buffers](#)

virtual (C# Reference)

12/28/2021 • 3 minutes to read • [Edit Online](#)

The `virtual` keyword is used to modify a method, property, indexer, or event declaration and allow for it to be overridden in a derived class. For example, this method can be overridden by any class that inherits it:

```
public virtual double Area()
{
    return x * y;
}
```

The implementation of a virtual member can be changed by an [overriding member](#) in a derived class. For more information about how to use the `virtual` keyword, see [Versioning with the Override and New Keywords](#) and [Knowing When to Use Override and New Keywords](#).

Remarks

When a virtual method is invoked, the run-time type of the object is checked for an overriding member. The overriding member in the most derived class is called, which might be the original member, if no derived class has overridden the member.

By default, methods are non-virtual. You cannot override a non-virtual method.

You cannot use the `virtual` modifier with the `static`, `abstract`, `private`, or `override` modifiers. The following example shows a virtual property:

```

class MyBaseClass
{
    // virtual auto-implemented property. Overrides can only
    // provide specialized behavior if they implement get and set accessors.
    public virtual string Name { get; set; }

    // ordinary virtual property with backing field
    private int _num;
    public virtual int Number
    {
        get { return _num; }
        set { _num = value; }
    }
}

class MyDerivedClass : MyBaseClass
{
    private string _name;

    // Override auto-implemented property with ordinary property
    // to provide specialized accessor behavior.
    public override string Name
    {
        get
        {
            return _name;
        }
        set
        {
            if (!string.IsNullOrEmpty(value))
            {
                _name = value;
            }
            else
            {
                _name = "Unknown";
            }
        }
    }
}

```

Virtual properties behave like virtual methods, except for the differences in declaration and invocation syntax.

- It is an error to use the `virtual` modifier on a static property.
- A virtual inherited property can be overridden in a derived class by including a property declaration that uses the `override` modifier.

Example

In this example, the `Shape` class contains the two coordinates `x`, `y`, and the `Area()` virtual method. Different shape classes such as `Circle`, `Cylinder`, and `Sphere` inherit the `Shape` class, and the surface area is calculated for each figure. Each derived class has its own override implementation of `Area()`.

Notice that the inherited classes `Circle`, `Sphere`, and `Cylinder` all use constructors that initialize the base class, as shown in the following declaration.

```

public Cylinder(double r, double h): base(r, h) {}

```

The following program calculates and displays the appropriate area for each figure by invoking the appropriate implementation of the `Area()` method, according to the object that is associated with the method.

```

class TestClass
{
    public class Shape
    {
        public const double PI = Math.PI;
        protected double _x, _y;

        public Shape()
        {
        }

        public Shape(double x, double y)
        {
            _x = x;
            _y = y;
        }

        public virtual double Area()
        {
            return _x * _y;
        }
    }

    public class Circle : Shape
    {
        public Circle(double r) : base(r, 0)
        {
        }

        public override double Area()
        {
            return PI * _x * _x;
        }
    }

    public class Sphere : Shape
    {
        public Sphere(double r) : base(r, 0)
        {
        }

        public override double Area()
        {
            return 4 * PI * _x * _x;
        }
    }

    public class Cylinder : Shape
    {
        public Cylinder(double r, double h) : base(r, h)
        {
        }

        public override double Area()
        {
            return 2 * PI * _x * _x + 2 * PI * _x * _y;
        }
    }

    static void Main()
    {
        double r = 3.0, h = 5.0;
        Shape c = new Circle(r);
        Shape s = new Sphere(r);
        Shape l = new Cylinder(r, h);
        // Display results.
        Console.WriteLine("Area of Circle    = {0:F2}", c.Area());
        Console.WriteLine("Area of Sphere    = {0:F2}", s.Area());
    }
}

```

```
        Console.WriteLine("Area of Cylinder = {0:F2}", l.Area());
    }
}
/*
Output:
Area of Circle   = 28.27
Area of Sphere   = 113.10
Area of Cylinder = 150.80
*/
```

C# language specification

For more information, see the [C# Language Specification](#). The language specification is the definitive source for C# syntax and usage.

See also

- [Polymorphism](#)
- [abstract](#)
- [override](#)
- [new \(modifier\)](#)

volatile (C# Reference)

12/28/2021 • 3 minutes to read • [Edit Online](#)

The `volatile` keyword indicates that a field might be modified by multiple threads that are executing at the same time. The compiler, the runtime system, and even hardware may rearrange reads and writes to memory locations for performance reasons. Fields that are declared `volatile` are excluded from certain kinds of optimizations. There is no guarantee of a single total ordering of volatile writes as seen from all threads of execution. For more information, see the [Volatile](#) class.

NOTE

On a multiprocessor system, a volatile read operation does not guarantee to obtain the latest value written to that memory location by any processor. Similarly, a volatile write operation does not guarantee that the value written would be immediately visible to other processors.

The `volatile` keyword can be applied to fields of these types:

- Reference types.
- Pointer types (in an unsafe context). Note that although the pointer itself can be volatile, the object that it points to cannot. In other words, you cannot declare a "pointer to volatile."
- Simple types such as `sbyte`, `byte`, `short`, `ushort`, `int`, `uint`, `char`, `float`, and `bool`.
- An `enum` type with one of the following base types: `byte`, `sbyte`, `short`, `ushort`, `int`, or `uint`.
- Generic type parameters known to be reference types.
- [IntPtr](#) and [UIntPtr](#).

Other types, including `double` and `long`, cannot be marked `volatile` because reads and writes to fields of those types cannot be guaranteed to be atomic. To protect multi-threaded access to those types of fields, use the [Interlocked](#) class members or protect access using the `lock` statement.

The `volatile` keyword can only be applied to fields of a `class` or `struct`. Local variables cannot be declared `volatile`.

Example

The following example shows how to declare a public field variable as `volatile`.

```
class VolatileTest
{
    public volatile int sharedStorage;

    public void Test(int i)
    {
        sharedStorage = i;
    }
}
```

The following example demonstrates how an auxiliary or worker thread can be created and used to perform processing in parallel with that of the primary thread. For more information about multithreading, see [Managed Threading](#).

```

public class Worker
{
    // This method is called when the thread is started.
    public void DoWork()
    {
        bool work = false;
        while (!_shouldStop)
        {
            work = !work; // simulate some work
        }
        Console.WriteLine("Worker thread: terminating gracefully.");
    }
    public void RequestStop()
    {
        _shouldStop = true;
    }
    // Keyword volatile is used as a hint to the compiler that this data
    // member is accessed by multiple threads.
    private volatile bool _shouldStop;
}

public class WorkerThreadExample
{
    public static void Main()
    {
        // Create the worker thread object. This does not start the thread.
        Worker workerObject = new Worker();
        Thread workerThread = new Thread(workerObject.DoWork);

        // Start the worker thread.
        workerThread.Start();
        Console.WriteLine("Main thread: starting worker thread...");

        // Loop until the worker thread activates.
        while (!workerThread.IsAlive)
            ;

        // Put the main thread to sleep for 500 milliseconds to
        // allow the worker thread to do some work.
        Thread.Sleep(500);

        // Request that the worker thread stop itself.
        workerObject.RequestStop();

        // Use the Thread.Join method to block the current thread
        // until the object's thread terminates.
        workerThread.Join();
        Console.WriteLine("Main thread: worker thread has terminated.");
    }
    // Sample output:
    // Main thread: starting worker thread...
    // Worker thread: terminating gracefully.
    // Main thread: worker thread has terminated.
}

```

With the `volatile` modifier added to the declaration of `_shouldStop` in place, you'll always get the same results (similar to the excerpt shown in the preceding code). However, without that modifier on the `_shouldStop` member, the behavior is unpredictable. The `DoWork` method may optimize the member access, resulting in reading stale data. Because of the nature of multi-threaded programming, the number of stale reads is unpredictable. Different runs of the program will produce somewhat different results.

C# language specification

For more information, see the [C# Language Specification](#). The language specification is the definitive source for

C# syntax and usage.

See also

- [C# language specification: volatile keyword](#)
- [C# Reference](#)
- [C# Programming Guide](#)
- [C# Keywords](#)
- [Modifiers](#)
- [lock statement](#)
- [Interlocked](#)

Statement keywords (C# Reference)

12/28/2021 • 2 minutes to read • [Edit Online](#)

Statements are program instructions. Except as described in the topics referenced in the following table, statements are executed in sequence. The following table lists the C# statement keywords. For more information about statements that are not expressed with any keyword, see [Statements](#).

CATEGORY	C# KEYWORDS
Selection statements	<code>if</code> , <code>switch</code>
Iteration statements	<code>do</code> , <code>for</code> , <code>foreach</code> , <code>while</code>
Jump statements	<code>break</code> , <code>continue</code> , <code>goto</code> , <code>return</code>
Exception handling statements	throw , try-catch , try-finally , try-catch-finally
Checked and unchecked	checked , unchecked
fixed statement	fixed
lock statement	<code>lock</code>
yield statement	<code>yield</code>

See also

- [C# Reference](#)
- [Statements](#)
- [C# Keywords](#)

throw (C# Reference)

12/28/2021 • 3 minutes to read • [Edit Online](#)

Signals the occurrence of an exception during program execution.

Remarks

The syntax of `throw` is:

```
throw [e];
```

where `e` is an instance of a class derived from [System.Exception](#). The following example uses the `throw` statement to throw an [IndexOutOfRangeException](#) if the argument passed to a method named `GetNumber` does not correspond to a valid index of an internal array.

```
using System;

namespace Throw2
{
    public class NumberGenerator
    {
        int[] numbers = { 2, 4, 6, 8, 10, 12, 14, 16, 18, 20 };

        public int GetNumber(int index)
        {
            if (index < 0 || index >= numbers.Length)
            {
                throw new IndexOutOfRangeException();
            }
            return numbers[index];
        }
    }
}
```

Method callers then use a `try-catch` or `try-catch-finally` block to handle the thrown exception. The following example handles the exception thrown by the `GetNumber` method.

```
using System;

public class Example
{
    public static void Main()
    {
        var gen = new NumberGenerator();
        int index = 10;
        try
        {
            int value = gen.GetNumber(index);
            Console.WriteLine($"Retrieved {value}");
        }
        catch (IndexOutOfRangeException e)
        {
            Console.WriteLine($"{e.GetType().Name}: {index} is outside the bounds of the array");
        }
    }
}

// The example displays the following output:
//      IndexOutOfRangeException: 10 is outside the bounds of the array
```

Re-throwing an exception

`throw` can also be used in a `catch` block to re-throw an exception handled in a `catch` block. In this case, `throw` does not take an exception operand. It is most useful when a method passes on an argument from a caller to some other library method, and the library method throws an exception that must be passed on to the caller. For example, the following example re-throws an [NullReferenceException](#) that is thrown when attempting to retrieve the first character of an uninitialized string.

```

using System;

namespace Throw
{
    public class Sentence
    {
        public Sentence(string s)
        {
            Value = s;
        }

        public string Value { get; set; }

        public char GetFirstCharacter()
        {
            try
            {
                return Value[0];
            }
            catch (NullReferenceException e)
            {
                throw;
            }
        }
    }

    public class Example
    {
        public static void Main()
        {
            var s = new Sentence(null);
            Console.WriteLine($"The first character is {s.GetFirstCharacter()}");
        }
    }
}
// The example displays the following output:
//      Unhandled Exception: System.NullReferenceException: Object reference not set to an instance of an
//      object.
//          at Sentence.GetFirstCharacter()
//          at Example.Main()

```

IMPORTANT

You can also use the `throw e` syntax in a `catch` block to instantiate a new exception that you pass on to the caller. In this case, the stack trace of the original exception, which is available from the [StackTrace](#) property, is not preserved.

The `throw` expression

Starting with C# 7.0, `throw` can be used as an expression as well as a statement. This allows an exception to be thrown in contexts that were previously unsupported. These include:

- [the conditional operator](#). The following example uses a `throw` expression to throw an [ArgumentException](#) if a method is passed an empty string array. Before C# 7.0, this logic would need to appear in an `if / else` statement.

```
private static void DisplayFirstNumber(string[] args)
{
    string arg = args.Length >= 1 ? args[0] :
        throw new ArgumentException("You must supply an argument");
    if (Int64.TryParse(arg, out var number))
        Console.WriteLine($"You entered {number:F0}");
    else
        Console.WriteLine($"{arg} is not a number.");
}
```

- [the null-coalescing operator](#). In the following example, a `throw` expression is used with a null-coalescing operator to throw an exception if the string assigned to a `Name` property is `null`.

```
public string Name
{
    get => name;
    set => name = value ??
        throw new ArgumentNullException(paramName: nameof(value), message: "Name cannot be null");
}
```

- an expression-bodied [lambda](#) or method. The following example illustrates an expression-bodied method that throws an [InvalidCastException](#) because a conversion to a [DateTime](#) value is not supported.

```
DateTime ToDateTime(IFormatProvider provider) =>
    throw new InvalidCastException("Conversion to a DateTime is not supported.");
```

C# language specification

For more information, see the [C# Language Specification](#). The language specification is the definitive source for C# syntax and usage.

See also

- [C# Reference](#)
- [C# Programming Guide](#)
- [try-catch](#)
- [C# Keywords](#)
- [How to: Explicitly Throw Exceptions](#)

try-catch (C# Reference)

12/28/2021 • 9 minutes to read • [Edit Online](#)

The try-catch statement consists of a `try` block followed by one or more `catch` clauses, which specify handlers for different exceptions.

When an exception is thrown, the common language runtime (CLR) looks for the `catch` statement that handles this exception. If the currently executing method does not contain such a `catch` block, the CLR looks at the method that called the current method, and so on up the call stack. If no `catch` block is found, then the CLR displays an unhandled exception message to the user and stops execution of the program.

The `try` block contains the guarded code that may cause the exception. The block is executed until an exception is thrown or it is completed successfully. For example, the following attempt to cast a `null` object raises the [NullReferenceException](#) exception:

```
object o2 = null;
try
{
    int i2 = (int)o2;    // Error
}
```

Although the `catch` clause can be used without arguments to catch any type of exception, this usage is not recommended. In general, you should only catch those exceptions that you know how to recover from. Therefore, you should always specify an object argument derived from [System.Exception](#). The exception type should be as specific as possible in order to avoid incorrectly accepting exceptions that your exception handler is actually not able to resolve. As such, prefer concrete exceptions over the base `Exception` type. For example:

```
catch (InvalidCastException e)
{
    // recover from exception
}
```

It is possible to use more than one specific `catch` clause in the same try-catch statement. In this case, the order of the `catch` clauses is important because the `catch` clauses are examined in order. Catch the more specific exceptions before the less specific ones. The compiler produces an error if you order your catch blocks so that a later block can never be reached.

Using `catch` arguments is one way to filter for the exceptions you want to handle. You can also use an exception filter that further examines the exception to decide whether to handle it. If the exception filter returns false, then the search for a handler continues.

```
catch (ArgumentException e) when (e.ParamName == "...")
{
    // recover from exception
}
```

Exception filters are preferable to catching and rethrowing (explained below) because filters leave the stack unharmed. If a later handler dumps the stack, you can see where the exception originally came from, rather than just the last place it was rethrown. A common use of exception filter expressions is logging. You can create a filter that always returns false that also outputs to a log, you can log exceptions as they go by without having to

handle them and rethrow.

A `throw` statement can be used in a `catch` block to re-throw the exception that is caught by the `catch` statement. The following example extracts source information from an `IOException` exception, and then throws the exception to the parent method.

```
catch (FileNotFoundException e)
{
    // FileNotFoundExceptions are handled here.
}
catch (IOException e)
{
    // Extract some information from this exception, and then
    // throw it to the parent method.
    if (e.Source != null)
        Console.WriteLine("IOException source: {0}", e.Source);
    throw;
}
```

You can catch one exception and throw a different exception. When you do this, specify the exception that you caught as the inner exception, as shown in the following example.

```
catch (InvalidCastException e)
{
    // Perform some action here, and then throw a new exception.
    throw new YourCustomException("Put your error message here.", e);
}
```

You can also re-throw an exception when a specified condition is true, as shown in the following example.

```
catch (InvalidCastException e)
{
    if (e.Data == null)
    {
        throw;
    }
    else
    {
        // Take some action.
    }
}
```

NOTE

It is also possible to use an exception filter to get a similar result in an often cleaner fashion (as well as not modifying the stack, as explained earlier in this document). The following example has a similar behavior for callers as the previous example. The function throws the `InvalidCastException` back to the caller when `e.Data` is `null`.

```
catch (InvalidCastException e) when (e.Data != null)
{
    // Take some action.
}
```

From inside a `try` block, initialize only variables that are declared therein. Otherwise, an exception can occur before the execution of the block is completed. For example, in the following code example, the variable `n` is initialized inside the `try` block. An attempt to use this variable outside the `try` block in the `Write(n)` statement will generate a compiler error.

```
static void Main()
{
    int n;
    try
    {
        // Do not initialize this variable here.
        n = 123;
    }
    catch
    {
    }
    // Error: Use of unassigned local variable 'n'.
    Console.Write(n);
}
```

For more information about catch, see [try-catch-finally](#).

Exceptions in async methods

An async method is marked by an [async](#) modifier and usually contains one or more await expressions or statements. An await expression applies the [await](#) operator to a [Task](#) or [Task<TResult>](#).

When control reaches an `await` in the async method, progress in the method is suspended until the awaited task completes. When the task is complete, execution can resume in the method. For more information, see [Asynchronous programming with async and await](#).

The completed task to which `await` is applied might be in a faulted state because of an unhandled exception in the method that returns the task. Awaiting the task throws an exception. A task can also end up in a canceled state if the asynchronous process that returns it is canceled. Awaiting a canceled task throws an `OperationCanceledException`.

To catch the exception, await the task in a `try` block, and catch the exception in the associated `catch` block. For an example, see the [Async method example](#) section.

A task can be in a faulted state because multiple exceptions occurred in the awaited async method. For example, the task might be the result of a call to [Task.WhenAll](#). When you await such a task, only one of the exceptions is caught, and you can't predict which exception will be caught. For an example, see the [Task.WhenAll example](#) section.

Example

In the following example, the `try` block contains a call to the `ProcessString` method that may cause an exception. The `catch` clause contains the exception handler that just displays a message on the screen. When the `throw` statement is called from inside `ProcessString`, the system looks for the `catch` statement and displays the message `Exception caught`.


```

class TryFinallyTest
{
    static void ProcessString(string s)
    {
        if (s == null)
        {
            throw new ArgumentNullException(paramName: nameof(s), message: "parameter can't be null.");
        }
    }

    public static void Main()
    {
        string s = null; // For demonstration purposes.

        try
        {
            ProcessString(s);
        }
        catch (Exception e)
        {
            Console.WriteLine("{0} Exception caught.", e);
        }
    }
}
/*
Output:
System.ArgumentNullException: Value cannot be null.
   at TryFinallyTest.Main() Exception caught.
* */

```

Two catch blocks example

In the following example, two catch blocks are used, and the most specific exception, which comes first, is caught.

To catch the least specific exception, you can replace the throw statement in `ProcessString` with the following statement: `throw new Exception()`.

If you place the least-specific catch block first in the example, the following error message appears:

```
A previous catch clause already catches all exceptions of this or a super type ('System.Exception').
```

```

class ThrowTest3
{
    static void ProcessString(string s)
    {
        if (s == null)
        {
            throw new ArgumentNullException(paramName: nameof(s), message: "Parameter can't be null");
        }
    }

    public static void Main()
    {
        try
        {
            string s = null;
            ProcessString(s);
        }
        // Most specific:
        catch (ArgumentNullException e)
        {
            Console.WriteLine("{0} First exception caught.", e);
        }
        // Least specific:
        catch (Exception e)
        {
            Console.WriteLine("{0} Second exception caught.", e);
        }
    }
}
/*
Output:
System.ArgumentNullException: Value cannot be null.
at Test.ThrowTest3.ProcessString(String s) ... First exception caught.
*/

```

Async method example

The following example illustrates exception handling for async methods. To catch an exception that an async task throws, place the `await` expression in a `try` block, and catch the exception in a `catch` block.

Uncomment the `throw new Exception` line in the example to demonstrate exception handling. The task's `IsFaulted` property is set to `True`, the task's `Exception.InnerException` property is set to the exception, and the exception is caught in the `catch` block.

Uncomment the `throw new OperationCanceledException` line to demonstrate what happens when you cancel an asynchronous process. The task's `IsCanceled` property is set to `true`, and the exception is caught in the `catch` block. Under some conditions that don't apply to this example, the task's `IsFaulted` property is set to `true` and `IsCanceled` is set to `false`.

```

public async Task DoSomethingAsync()
{
    Task<string> theTask = DelayAsync();

    try
    {
        string result = await theTask;
        Debug.WriteLine("Result: " + result);
    }
    catch (Exception ex)
    {
        Debug.WriteLine("Exception Message: " + ex.Message);
    }
    Debug.WriteLine("Task IsCanceled: " + theTask.IsCanceled);
    Debug.WriteLine("Task IsFaulted: " + theTask.IsFaulted);
    if (theTask.Exception != null)
    {
        Debug.WriteLine("Task Exception Message: "
            + theTask.Exception.Message);
        Debug.WriteLine("Task Inner Exception Message: "
            + theTask.Exception.InnerException.Message);
    }
}

private async Task<string> DelayAsync()
{
    await Task.Delay(100);

    // Uncomment each of the following lines to
    // demonstrate exception handling.

    //throw new OperationCanceledException("canceled");
    //throw new Exception("Something happened.");
    return "Done";
}

// Output when no exception is thrown in the awaited method:
// Result: Done
// Task IsCanceled: False
// Task IsFaulted: False

// Output when an Exception is thrown in the awaited method:
// Exception Message: Something happened.
// Task IsCanceled: False
// Task IsFaulted: True
// Task Exception Message: One or more errors occurred.
// Task Inner Exception Message: Something happened.

// Output when a OperationCanceledException or TaskCanceledException
// is thrown in the awaited method:
// Exception Message: canceled
// Task IsCanceled: True
// Task IsFaulted: False

```

Task.WhenAll example

The following example illustrates exception handling where multiple tasks can result in multiple exceptions. The `try` block awaits the task that's returned by a call to [Task.WhenAll](#). The task is complete when the three tasks to which `WhenAll` is applied are complete.

Each of the three tasks causes an exception. The `catch` block iterates through the exceptions, which are found in the `Exception.InnerExceptions` property of the task that was returned by [Task.WhenAll](#).

```

public async Task DoMultipleAsync()
{
    Task theTask1 = ExcAsync(info: "First Task");
    Task theTask2 = ExcAsync(info: "Second Task");
    Task theTask3 = ExcAsync(info: "Third Task");

    Task allTasks = Task.WhenAll(theTask1, theTask2, theTask3);

    try
    {
        await allTasks;
    }
    catch (Exception ex)
    {
        Debug.WriteLine("Exception: " + ex.Message);
        Debug.WriteLine("Task IsFaulted: " + allTasks.IsFaulted);
        foreach (var inEx in allTasks.Exception.InnerExceptions)
        {
            Debug.WriteLine("Task Inner Exception: " + inEx.Message);
        }
    }
}

private async Task ExcAsync(string info)
{
    await Task.Delay(100);

    throw new Exception("Error-" + info);
}

// Output:
// Exception: Error-First Task
// Task IsFaulted: True
// Task Inner Exception: Error-First Task
// Task Inner Exception: Error-Second Task
// Task Inner Exception: Error-Third Task

```

C# language specification

For more information, see [The try statement](#) section of the [C# language specification](#).

See also

- [C# Reference](#)
- [C# Programming Guide](#)
- [C# Keywords](#)
- [try, throw, and catch Statements \(C++\)](#)
- [throw](#)
- [try-finally](#)
- [How to: Explicitly Throw Exceptions](#)

try-finally (C# Reference)

12/28/2021 • 3 minutes to read • [Edit Online](#)

By using a `finally` block, you can clean up any resources that are allocated in a `try` block, and you can run code even if an exception occurs in the `try` block. Typically, the statements of a `finally` block run when control leaves a `try` statement. The transfer of control can occur as a result of normal execution, of execution of a `break`, `continue`, `goto`, or `return` statement, or of propagation of an exception out of the `try` statement.

Within a handled exception, the associated `finally` block is guaranteed to be run. However, if the exception is unhandled, execution of the `finally` block is dependent on how the exception unwind operation is triggered. That, in turn, is dependent on how your computer is set up. The only cases where `finally` clauses don't run involve a program being immediately stopped. An example of this would be when `InvalidProgramException` gets thrown because of the IL statements being corrupt. On most operating systems, reasonable resource cleanup will take place as part of stopping and unloading the process.

Usually, when an unhandled exception ends an application, whether or not the `finally` block is run is not important. However, if you have statements in a `finally` block that must be run even in that situation, one solution is to add a `catch` block to the `try` - `finally` statement. Alternatively, you can catch the exception that might be thrown in the `try` block of a `try` - `finally` statement higher up the call stack. That is, you can catch the exception in the method that calls the method that contains the `try` - `finally` statement, or in the method that calls that method, or in any method in the call stack. If the exception is not caught, execution of the `finally` block depends on whether the operating system chooses to trigger an exception unwind operation.

Example

In the following example, an invalid conversion statement causes a `System.InvalidCastException` exception. The exception is unhandled.

```

public class ThrowTestA
{
    public static void Main()
    {
        int i = 123;
        string s = "Some string";
        object obj = s;

        try
        {
            // Invalid conversion; obj contains a string, not a numeric type.
            i = (int)obj;

            // The following statement is not run.
            Console.WriteLine("WriteLine at the end of the try block.");
        }
        finally
        {
            // To run the program in Visual Studio, type CTRL+F5. Then
            // click Cancel in the error dialog.
            Console.WriteLine("\nExecution of the finally block after an unhandled\n" +
                "error depends on how the exception unwind operation is triggered.");
            Console.WriteLine("i = {0}", i);
        }
    }
}
// Output:
// Unhandled Exception: System.InvalidCastException: Specified cast is not valid.
//
// Execution of the finally block after an unhandled
// error depends on how the exception unwind operation is triggered.
// i = 123
}

```

In the following example, an exception from the `TryCast` method is caught in a method farther up the call stack.

```

public class ThrowTestB
{
    public static void Main()
    {
        try
        {
            // TryCast produces an unhandled exception.
            TryCast();
        }
        catch (Exception ex)
        {
            // Catch the exception that is unhandled in TryCast.
            Console.WriteLine
                ("Catching the {0} exception triggers the finally block.",
                 ex.GetType());

            // Restore the original unhandled exception. You might not
            // know what exception to expect, or how to handle it, so pass
            // it on.
            throw;
        }
    }

    static void TryCast()
    {
        int i = 123;
        string s = "Some string";
        object obj = s;

        try
        {
            // Invalid conversion; obj contains a string, not a numeric type.
            i = (int)obj;

            // The following statement is not run.
            Console.WriteLine("WriteLine at the end of the try block.");
        }
        finally
        {
            // Report that the finally block is run, and show that the value of
            // i has not been changed.
            Console.WriteLine("\nIn the finally block in TryCast, i = {0}.\n", i);
        }
    }
}
// Output:
// In the finally block in TryCast, i = 123.

// Catching the System.InvalidCastException exception triggers the finally block.

// Unhandled Exception: System.InvalidCastException: Specified cast is not valid.
}

```

For more information about `finally`, see [try-catch-finally](#).

C# also contains the [using statement](#), which provides similar functionality for [IDisposable](#) objects in a convenient syntax.

C# language specification

For more information, see [The try statement](#) section of the [C# language specification](#).

See also

- [C# Reference](#)

- [C# Programming Guide](#)
- [C# Keywords](#)
- [try, throw, and catch Statements \(C++\)](#)
- [throw](#)
- [try-catch](#)
- [How to: Explicitly Throw Exceptions](#)

try-catch-finally (C# Reference)

12/28/2021 • 2 minutes to read • [Edit Online](#)

A common usage of `catch` and `finally` together is to obtain and use resources in a `try` block, deal with exceptional circumstances in a `catch` block, and release the resources in the `finally` block.

For more information and examples on re-throwing exceptions, see [try-catch](#) and [Throwing Exceptions](#). For more information about the `finally` block, see [try-finally](#).

Example

```
public class EHClass
{
    void ReadFile(int index)
    {
        // To run this code, substitute a valid path from your local machine
        string path = @"c:\users\public\test.txt";
        System.IO.StreamReader file = new System.IO.StreamReader(path);
        char[] buffer = new char[10];
        try
        {
            file.ReadBlock(buffer, index, buffer.Length);
        }
        catch (System.IO.IOException e)
        {
            Console.WriteLine("Error reading from {0}. Message = {1}", path, e.Message);
        }
        finally
        {
            if (file != null)
            {
                file.Close();
            }
        }
        // Do something with buffer...
    }
}
```

C# language specification

For more information, see [The try statement](#) section of the [C# language specification](#).

See also

- [C# Reference](#)
- [C# Programming Guide](#)
- [C# Keywords](#)
- [try, throw, and catch Statements \(C++\)](#)
- [throw](#)
- [How to: Explicitly Throw Exceptions](#)
- [using Statement](#)

Checked and Unchecked (C# Reference)

12/28/2021 • 2 minutes to read • [Edit Online](#)

C# statements can execute in either checked or unchecked context. In a checked context, arithmetic overflow raises an exception. In an unchecked context, arithmetic overflow is ignored and the result is truncated by discarding any high-order bits that don't fit in the destination type.

- [checked](#) Specify checked context.
- [unchecked](#) Specify unchecked context.

The following operations are affected by the overflow checking:

- Expressions using the following predefined operators on integral types:

`++`, `--`, unary `-`, `+`, `-`, `*`, `/`

- Explicit numeric conversions between integral types, or from `float` or `double` to an integral type.

If neither `checked` nor `unchecked` is specified, the default context for non-constant expressions (expressions that are evaluated at run time) is defined by the value of the [CheckForOverflowUnderflow](#) compiler option. By default the value of that option is unset and arithmetic operations are executed in an unchecked context.

For constant expressions (expressions that can be fully evaluated at compile time), the default context is always checked. Unless a constant expression is explicitly placed in an unchecked context, overflows that occur during the compile-time evaluation of the expression cause compile-time errors.

See also

- [C# Reference](#)
- [C# Programming Guide](#)
- [C# Keywords](#)
- [Statement Keywords](#)

checked (C# Reference)

12/28/2021 • 2 minutes to read • [Edit Online](#)

The `checked` keyword is used to explicitly enable overflow checking for integral-type arithmetic operations and conversions.

By default, an expression that contains only constant values causes a compiler error if the expression produces a value that is outside the range of the destination type. If the expression contains one or more non-constant values, the compiler does not detect the overflow. Evaluating the expression assigned to `i2` in the following example does not cause a compiler error.

```
// The following example causes compiler error CS0220 because 2147483647
// is the maximum value for integers.
//int i1 = 2147483647 + 10;

// The following example, which includes variable ten, does not cause
// a compiler error.
int ten = 10;
int i2 = 2147483647 + ten;

// By default, the overflow in the previous statement also does
// not cause a run-time exception. The following line displays
// -2,147,483,639 as the sum of 2,147,483,647 and 10.
Console.WriteLine(i2);
```

By default, these non-constant expressions are not checked for overflow at run time either, and they do not raise overflow exceptions. The previous example displays -2,147,483,639 as the sum of two positive integers.

Overflow checking can be enabled by compiler options, environment configuration, or use of the `checked` keyword. The following examples demonstrate how to use a `checked` expression or a `checked` block to detect the overflow that is produced by the previous sum at run time. Both examples raise an overflow exception.

```
// If the previous sum is attempted in a checked environment, an
// OverflowException error is raised.

// Checked expression.
Console.WriteLine(checked(2147483647 + ten));

// Checked block.
checked
{
    int i3 = 2147483647 + ten;
    Console.WriteLine(i3);
}
```

The `unchecked` keyword can be used to prevent overflow checking.

Example

This sample shows how to use `checked` to enable overflow checking at run time.

```

class OverFlowTest
{
    // Set maxIntValue to the maximum value for integers.
    static int maxIntValue = 2147483647;

    // Using a checked expression.
    static int CheckedMethod()
    {
        int z = 0;
        try
        {
            // The following line raises an exception because it is checked.
            z = checked(maxIntValue + 10);
        }
        catch (System.OverflowException e)
        {
            // The following line displays information about the error.
            Console.WriteLine("CHECKED and CAUGHT: " + e.ToString());
        }
        // The value of z is still 0.
        return z;
    }

    // Using an unchecked expression.
    static int UncheckedMethod()
    {
        int z = 0;
        try
        {
            // The following calculation is unchecked and will not
            // raise an exception.
            z = maxIntValue + 10;
        }
        catch (System.OverflowException e)
        {
            // The following line will not be executed.
            Console.WriteLine("UNCHECKED and CAUGHT: " + e.ToString());
        }
        // Because of the undetected overflow, the sum of 2147483647 + 10 is
        // returned as -2147483639.
        return z;
    }

    static void Main()
    {
        Console.WriteLine("\nCHECKED output value is: {0}",
            CheckedMethod());
        Console.WriteLine("UNCHECKED output value is: {0}",
            UncheckedMethod());
    }
}
/*
Output:
CHECKED and CAUGHT: System.OverflowException: Arithmetic operation resulted
in an overflow.
    at ConsoleApplication1.OverFlowTest.CheckedMethod()

CHECKED output value is: 0
UNCHECKED output value is: -2147483639
*/
}

```

C# language specification

For more information, see the [C# Language Specification](#). The language specification is the definitive source for C# syntax and usage.

See also

- [C# Reference](#)
- [C# Programming Guide](#)
- [C# Keywords](#)
- [Checked and Unchecked](#)
- [unchecked](#)

unchecked (C# Reference)

12/28/2021 • 2 minutes to read • [Edit Online](#)

The `unchecked` keyword is used to suppress overflow-checking for integral-type arithmetic operations and conversions.

In an unchecked context, if an expression produces a value that is outside the range of the destination type, the overflow is not flagged. For example, because the calculation in the following example is performed in an `unchecked` block or expression, the fact that the result is too large for an integer is ignored, and `int1` is assigned the value -2,147,483,639.

```
unchecked
{
    int1 = 2147483647 + 10;
}
int1 = unchecked(ConstantMax + 10);
```

If the `unchecked` environment is removed, a compilation error occurs. The overflow can be detected at compile time because all the terms of the expression are constants.

Expressions that contain non-constant terms are unchecked by default at compile time and run time. See [checked](#) for information about enabling a checked environment.

Because checking for overflow takes time, the use of unchecked code in situations where there is no danger of overflow might improve performance. However, if overflow is a possibility, a checked environment should be used.

Example

This sample shows how to use the `unchecked` keyword.

```

class UncheckedDemo
{
    static void Main(string[] args)
    {
        // int.MaxValue is 2,147,483,647.
        const int ConstantMax = int.MaxValue;
        int int1;
        int int2;
        int variableMax = 2147483647;

        // The following statements are checked by default at compile time. They do not
        // compile.
        //int1 = 2147483647 + 10;
        //int1 = ConstantMax + 10;

        // To enable the assignments to int1 to compile and run, place them inside
        // an unchecked block or expression. The following statements compile and
        // run.
        unchecked
        {
            int1 = 2147483647 + 10;
        }
        int1 = unchecked(ConstantMax + 10);

        // The sum of 2,147,483,647 and 10 is displayed as -2,147,483,639.
        Console.WriteLine(int1);

        // The following statement is unchecked by default at compile time and run
        // time because the expression contains the variable variableMax. It causes
        // overflow but the overflow is not detected. The statement compiles and runs.
        int2 = variableMax + 10;

        // Again, the sum of 2,147,483,647 and 10 is displayed as -2,147,483,639.
        Console.WriteLine(int2);

        // To catch the overflow in the assignment to int2 at run time, put the
        // declaration in a checked block or expression. The following
        // statements compile but raise an overflow exception at run time.
        checked
        {
            //int2 = variableMax + 10;
        }
        //int2 = checked(variableMax + 10);

        // Unchecked sections frequently are used to break out of a checked
        // environment in order to improve performance in a portion of code
        // that is not expected to raise overflow exceptions.
        checked
        {
            // Code that might cause overflow should be executed in a checked
            // environment.
            unchecked
            {
                // This section is appropriate for code that you are confident
                // will not result in overflow, and for which performance is
                // a priority.
            }
            // Additional checked code here.
        }
    }
}

```

C# language specification

For more information, see the [C# Language Specification](#). The language specification is the definitive source for

C# syntax and usage.

See also

- [C# Reference](#)
- [C# Programming Guide](#)
- [C# Keywords](#)
- [Checked and Unchecked](#)
- [checked](#)

fixed Statement (C# Reference)

12/28/2021 • 3 minutes to read • [Edit Online](#)

The `fixed` statement prevents the garbage collector from relocating a movable variable. The `fixed` statement is only permitted in an `unsafe` context. You can also use the `fixed` keyword to create [fixed size buffers](#).

The `fixed` statement sets a pointer to a managed variable and "pins" that variable during the execution of the statement. Pointers to movable managed variables are useful only in a `fixed` context. Without a `fixed` context, garbage collection could relocate the variables unpredictably. The C# compiler only lets you assign a pointer to a managed variable in a `fixed` statement.

```
class Point
{
    public int x;
    public int y;
}

unsafe private static void ModifyFixedStorage()
{
    // Variable pt is a managed variable, subject to garbage collection.
    Point pt = new Point();

    // Using fixed allows the address of pt members to be taken,
    // and "pins" pt so that it is not relocated.

    fixed (int* p = &pt.x)
    {
        *p = 1;
    }
}
```

You can initialize a pointer by using an array, a string, a fixed-size buffer, or the address of a variable. The following example illustrates the use of variable addresses, arrays, and strings:

```
Point point = new Point();
double[] arr = { 0, 1.5, 2.3, 3.4, 4.0, 5.9 };
string str = "Hello World";

// The following two assignments are equivalent. Each assigns the address
// of the first element in array arr to pointer p.

// You can initialize a pointer by using an array.
fixed (double* p = arr) { /*...*/ }

// You can initialize a pointer by using the address of a variable.
fixed (double* p = &arr[0]) { /*...*/ }

// The following assignment initializes p by using a string.
fixed (char* p = str) { /*...*/ }

// The following assignment is not valid, because str[0] is a char,
// which is a value, not a variable.
//fixed (char* p = &str[0]) { /*...*/ }
```

Starting with C# 7.3, the `fixed` statement operates on additional types beyond arrays, strings, fixed size buffers, or unmanaged variables. Any type that implements a method named `GetPinnableReference` can be pinned. The

`GetPinnableReference` must return a `ref` variable of an [unmanaged type](#). The .NET types [System.Span<T>](#) and [System.ReadOnlySpan<T>](#) introduced in .NET Core 2.0 make use of this pattern and can be pinned. This is shown in the following example:

```
unsafe private static void FixedSpanExample()
{
    int[] PascalsTriangle = {
        1,
        1, 1,
        1, 2, 1,
        1, 3, 3, 1,
        1, 4, 6, 4, 1,
        1, 5, 10, 10, 5, 1
    };

    Span<int> RowFive = new Span<int>(PascalsTriangle, 10, 5);

    fixed (int* ptrToRow = RowFive)
    {
        // Sum the numbers 1,4,6,4,1
        var sum = 0;
        for (int i = 0; i < RowFive.Length; i++)
        {
            sum += *(ptrToRow + i);
        }
        Console.WriteLine(sum);
    }
}
```

If you are creating types that should participate in this pattern, see [Span<T>.GetPinnableReference\(\)](#) for an example of implementing the pattern.

Multiple pointers can be initialized in one statement if they are all the same type:

```
fixed (byte* ps = srcarray, pd = dstarray) {...}
```

To initialize pointers of different types, simply nest `fixed` statements, as shown in the following example.

```
fixed (int* p1 = &point.x)
{
    fixed (double* p2 = &arr[5])
    {
        // Do something with p1 and p2.
    }
}
```

After the code in the statement is executed, any pinned variables are unpinned and subject to garbage collection. Therefore, do not point to those variables outside the `fixed` statement. The variables declared in the `fixed` statement are scoped to that statement, making this easier:

```
fixed (byte* ps = srcarray, pd = dstarray)
{
    ...
}
// ps and pd are no longer in scope here.
```

Pointers initialized in `fixed` statements are readonly variables. If you want to modify the pointer value, you must declare a second pointer variable, and modify that. The variable declared in the `fixed` statement cannot be modified:

```
fixed (byte* ps = srcarray, pd = dstarray)
{
    byte* pSourceCopy = ps;
    pSourceCopy++; // point to the next element.
    ps++; // invalid: cannot modify ps, as it is declared in the fixed statement.
}
```

You can allocate memory on the stack, where it is not subject to garbage collection and therefore does not need to be pinned. To do that, use a `stackalloc` expression.

C# language specification

For more information, see [The fixed statement](#) section of the [C# language specification](#).

See also

- [C# Reference](#)
- [C# Programming Guide](#)
- [C# Keywords](#)
- [unsafe](#)
- [Pointer types](#)
- [Fixed Size Buffers](#)

Method Parameters (C# Reference)

12/28/2021 • 2 minutes to read • [Edit Online](#)

Parameters declared for a method without [in](#), [ref](#) or [out](#), are passed to the called method by value. That value can be changed in the method, but the changed value will not be retained when control passes back to the calling procedure. By using a method parameter keyword, you can change this behavior.

This section describes the keywords you can use when declaring method parameters:

- [params](#) specifies that this parameter may take a variable number of arguments.
- [in](#) specifies that this parameter is passed by reference but is only read by the called method.
- [ref](#) specifies that this parameter is passed by reference and may be read or written by the called method.
- [out](#) specifies that this parameter is passed by reference and is written by the called method.

See also

- [C# Reference](#)
- [C# Programming Guide](#)
- [C# Keywords](#)

params (C# Reference)

12/28/2021 • 2 minutes to read • [Edit Online](#)

By using the `params` keyword, you can specify a [method parameter](#) that takes a variable number of arguments. The parameter type must be a single-dimensional array.

No additional parameters are permitted after the `params` keyword in a method declaration, and only one `params` keyword is permitted in a method declaration.

If the declared type of the `params` parameter is not a single-dimensional array, compiler error [CS0225](#) occurs.

When you call a method with a `params` parameter, you can pass in:

- A comma-separated list of arguments of the type of the array elements.
- An array of arguments of the specified type.
- No arguments. If you send no arguments, the length of the `params` list is zero.

Example

The following example demonstrates various ways in which arguments can be sent to a `params` parameter.

```

public class MyClass
{
    public static void UseParams(params int[] list)
    {
        for (int i = 0; i < list.Length; i++)
        {
            Console.Write(list[i] + " ");
        }
        Console.WriteLine();
    }

    public static void UseParams2(params object[] list)
    {
        for (int i = 0; i < list.Length; i++)
        {
            Console.Write(list[i] + " ");
        }
        Console.WriteLine();
    }

    static void Main()
    {
        // You can send a comma-separated list of arguments of the
        // specified type.
        UseParams(1, 2, 3, 4);
        UseParams2(1, 'a', "test");

        // A params parameter accepts zero or more arguments.
        // The following calling statement displays only a blank line.
        UseParams2();

        // An array argument can be passed, as long as the array
        // type matches the parameter type of the method being called.
        int[] myIntArray = { 5, 6, 7, 8, 9 };
        UseParams(myIntArray);

        object[] myObjArray = { 2, 'b', "test", "again" };
        UseParams2(myObjArray);

        // The following call causes a compiler error because the object
        // array cannot be converted into an integer array.
        //UseParams(myObjArray);

        // The following call does not cause an error, but the entire
        // integer array becomes the first element of the params array.
        UseParams2(myIntArray);
    }
}
/*
Output:
1 2 3 4
1 a test

5 6 7 8 9
2 b test again
System.Int32[]
*/

```

C# language specification

For more information, see the [C# Language Specification](#). The language specification is the definitive source for C# syntax and usage.

See also

- [C# Reference](#)
- [C# Programming Guide](#)
- [C# Keywords](#)
- [Method Parameters](#)

in parameter modifier (C# Reference)

12/28/2021 • 5 minutes to read • [Edit Online](#)

The `in` keyword causes arguments to be passed by reference but ensures the argument is not modified. It makes the formal parameter an alias for the argument, which must be a variable. In other words, any operation on the parameter is made on the argument. It is like the `ref` or `out` keywords, except that `in` arguments cannot be modified by the called method. Whereas `ref` arguments may be modified, `out` arguments must be modified by the called method, and those modifications are observable in the calling context.

```
int readonlyArgument = 44;
InArgExample(readonlyArgument);
Console.WriteLine(readonlyArgument);    // value is still 44

void InArgExample(in int number)
{
    // Uncomment the following line to see error CS8331
    //number = 19;
}
```

The preceding example demonstrates that the `in` modifier is usually unnecessary at the call site. It is only required in the method declaration.

NOTE

The `in` keyword can also be used with a generic type parameter to specify that the type parameter is contravariant, as part of a `foreach` statement, or as part of a `join` clause in a LINQ query. For more information on the use of the `in` keyword in these contexts, see [in](#), which provides links to all those uses.

Variables passed as `in` arguments must be initialized before being passed in a method call. However, the called method may not assign a value or modify the argument.

The `in` parameter modifier is available in C# 7.2 and later. Previous versions generate compiler error `CS8107` ("Feature 'readonly references' is not available in C# 7.0. Please use language version 7.2 or greater.") To configure the compiler language version, see [Select the C# language version](#).

Although `in`, `out`, and `ref` parameter modifiers are considered part of a signature, members declared in a single type cannot differ in signature solely by `in`, `ref` and `out`. Therefore, methods cannot be overloaded if the only difference is that one method takes a `ref` or `in` argument and the other takes an `out` argument. The following code, for example, will not compile:

```
class CS0663_Example
{
    // Compiler error CS0663: "Cannot define overloaded
    // methods that differ only on in, ref and out".
    public void SampleMethod(in int i) { }
    public void SampleMethod(ref int i) { }
}
```

Overloading based on the presence of `in` is allowed:


```
class InOverloads
{
    public void SampleMethod(in int i) { }
    public void SampleMethod(int i) { }
}
```

Overload resolution rules

You can understand the overload resolution rules for methods with by value vs. `in` arguments by understanding the motivation for `in` arguments. Defining methods using `in` parameters is a potential performance optimization. Some `struct` type arguments may be large in size, and when methods are called in tight loops or critical code paths, the cost of copying those structures is critical. Methods declare `in` parameters to specify that arguments may be passed by reference safely because the called method does not modify the state of that argument. Passing those arguments by reference avoids the (potentially) expensive copy.

Specifying `in` for arguments at the call site is typically optional. There is no semantic difference between passing arguments by value and passing them by reference using the `in` modifier. The `in` modifier at the call site is optional because you don't need to indicate that the argument's value might be changed. You explicitly add the `in` modifier at the call site to ensure the argument is passed by reference, not by value. Explicitly using `in` has the following two effects:

First, specifying `in` at the call site forces the compiler to select a method defined with a matching `in` parameter. Otherwise, when two methods differ only in the presence of `in`, the by value overload is a better match.

Second, specifying `in` declares your intent to pass an argument by reference. The argument used with `in` must represent a location that can be directly referred to. The same general rules for `out` and `ref` arguments apply: You cannot use constants, ordinary properties, or other expressions that produce values. Otherwise, omitting `in` at the call site informs the compiler that you will allow it to create a temporary variable to pass by read-only reference to the method. The compiler creates a temporary variable to overcome several restrictions with `in` arguments:

- A temporary variable allows compile-time constants as `in` parameters.
- A temporary variable allows properties, or other expressions for `in` parameters.
- A temporary variable allows arguments where there is an implicit conversion from the argument type to the parameter type.

In all the preceding instances, the compiler creates a temporary variable that stores the value of the constant, property, or other expression.

The following code illustrates these rules:

```
static void Method(in int argument)
{
    // implementation removed
}

Method(5); // OK, temporary variable created.
Method(5L); // CS1503: no implicit conversion from long to int
short s = 0;
Method(s); // OK, temporary int created with the value 0
Method(in s); // CS1503: cannot convert from in short to in int
int i = 42;
Method(i); // passed by readonly reference
Method(in i); // passed by readonly reference, explicitly using `in`
```

Now, suppose another method using by value arguments was available. The results change as shown in the following code:

```
static void Method(int argument)
{
    // implementation removed
}

static void Method(in int argument)
{
    // implementation removed
}

Method(5); // Calls overload passed by value
Method(5L); // CS1503: no implicit conversion from long to int
short s = 0;
Method(s); // Calls overload passed by value.
Method(in s); // CS1503: cannot convert from in short to in int
int i = 42;
Method(i); // Calls overload passed by value
Method(in i); // passed by readonly reference, explicitly using `in`
```

The only method call where the argument is passed by reference is the final one.

NOTE

The preceding code uses `int` as the argument type for simplicity. Because `int` is no larger than a reference in most modern machines, there is no benefit to passing a single `int` as a readonly reference.

Limitations on `in` parameters

You can't use the `in`, `ref`, and `out` keywords for the following kinds of methods:

- Async methods, which you define by using the `async` modifier.
- Iterator methods, which include a `yield return` or `yield break` statement.
- The first argument of an extension method cannot have the `in` modifier unless that argument is a struct.
- The first argument of an extension method where that argument is a generic type (even when that type is constrained to be a struct.)

You can learn more about the `in` modifier, how it differs from `ref` and `out` in the article on [Write safe efficient code](#).

C# Language Specification

For more information, see the [C# Language Specification](#). The language specification is the definitive source for C# syntax and usage.

ref (C# Reference)

12/28/2021 • 9 minutes to read • [Edit Online](#)

The `ref` keyword indicates that a value is passed by reference. It is used in four different contexts:

- In a method signature and in a method call, to pass an argument to a method by reference. For more information, see [Passing an argument by reference](#).
- In a method signature, to return a value to the caller by reference. For more information, see [Reference return values](#).
- In a member body, to indicate that a reference return value is stored locally as a reference that the caller intends to modify. Or to indicate that a local variable accesses another value by reference. For more information, see [Ref locals](#).
- In a `struct` declaration, to declare a `ref struct` or a `readonly ref struct`. For more information, see the [ref struct](#) section of the [Structure types](#) article.

Passing an argument by reference

When used in a method's parameter list, the `ref` keyword indicates that an argument is passed by reference, not by value. The `ref` keyword makes the formal parameter an alias for the argument, which must be a variable. In other words, any operation on the parameter is made on the argument.

For example, suppose the caller passes a local variable expression or an array element access expression. The called method can then replace the object to which the `ref` parameter refers. In that case, the caller's local variable or the array element refers to the new object when the method returns.

NOTE

Don't confuse the concept of passing by reference with the concept of reference types. The two concepts are not the same. A method parameter can be modified by `ref` regardless of whether it is a value type or a reference type. There is no boxing of a value type when it is passed by reference.

To use a `ref` parameter, both the method definition and the calling method must explicitly use the `ref` keyword, as shown in the following example. (Except that the calling method can omit `ref` when making a COM call.)

```
void Method(ref int refArgument)
{
    refArgument = refArgument + 44;
}

int number = 1;
Method(ref number);
Console.WriteLine(number);
// Output: 45
```

An argument that is passed to a `ref` or `in` parameter must be initialized before it is passed. This requirement differs from `out` parameters, whose arguments don't have to be explicitly initialized before they are passed.

Members of a class can't have signatures that differ only by `ref`, `in`, or `out`. A compiler error occurs if the only difference between two members of a type is that one of them has a `ref` parameter and the other has an `out`, or `in` parameter. The following code, for example, doesn't compile.

```
class CS0663_Example
{
    // Compiler error CS0663: "Cannot define overloaded
    // methods that differ only on ref and out".
    public void SampleMethod(out int i) { }
    public void SampleMethod(ref int i) { }
}
```

However, methods can be overloaded when one method has a `ref`, `in`, or `out` parameter and the other has a parameter that is passed by value, as shown in the following example.

```
class RefOverloadExample
{
    public void SampleMethod(int i) { }
    public void SampleMethod(ref int i) { }
}
```

In other situations that require signature matching, such as hiding or overriding, `in`, `ref`, and `out` are part of the signature and don't match each other.

Properties are not variables. They're methods, and cannot be passed to `ref` parameters.

You can't use the `ref`, `in`, and `out` keywords for the following kinds of methods:

- Async methods, which you define by using the [async](#) modifier.
- Iterator methods, which include a [yield return](#) or `yield break` statement.

[extension methods](#) also have restrictions on the use of these keywords:

- The `out` keyword cannot be used on the first argument of an extension method.
- The `ref` keyword cannot be used on the first argument of an extension method when the argument is not a struct, or a generic type not constrained to be a struct.
- The `in` keyword cannot be used unless the first argument is a struct. The `in` keyword cannot be used on any generic type, even when constrained to be a struct.

Passing an argument by reference: An example

The previous examples pass value types by reference. You can also use the `ref` keyword to pass reference types by reference. Passing a reference type by reference enables the called method to replace the object to which the reference parameter refers in the caller. The storage location of the object is passed to the method as the value of the reference parameter. If you change the value in the storage location of the parameter (to point to a new object), you also change the storage location to which the caller refers. The following example passes an instance of a reference type as a `ref` parameter.

```

class Product
{
    public Product(string name, int newID)
    {
        ItemName = name;
        ItemID = newID;
    }

    public string ItemName { get; set; }
    public int ItemID { get; set; }
}

private static void ChangeByReference(ref Product itemRef)
{
    // Change the address that is stored in the itemRef parameter.
    itemRef = new Product("Stapler", 99999);

    // You can change the value of one of the properties of
    // itemRef. The change happens to item in Main as well.
    itemRef.ItemID = 12345;
}

private static void ModifyProductsByReference()
{
    // Declare an instance of Product and display its initial values.
    Product item = new Product("Fasteners", 54321);
    System.Console.WriteLine("Original values in Main. Name: {0}, ID: {1}\n",
        item.ItemName, item.ItemID);

    // Pass the product instance to ChangeByReference.
    ChangeByReference(ref item);
    System.Console.WriteLine("Back in Main. Name: {0}, ID: {1}\n",
        item.ItemName, item.ItemID);
}

// This method displays the following output:
// Original values in Main. Name: Fasteners, ID: 54321
// Back in Main. Name: Stapler, ID: 12345

```

For more information about how to pass reference types by value and by reference, see [Passing Reference-Type Parameters](#).

Reference return values

Reference return values (or `ref` returns) are values that a method returns by reference to the caller. That is, the caller can modify the value returned by a method, and that change is reflected in the state of the object in the calling method.

A reference return value is defined by using the `ref` keyword:

- In the method signature. For example, the following method signature indicates that the `GetCurrentPrice` method returns a `Decimal` value by reference.

```
public ref decimal GetCurrentPrice()
```

- Between the `return` token and the variable returned in a `return` statement in the method. For example:

```
return ref DecimalArray[0];
```

In order for the caller to modify the object's state, the reference return value must be stored to a variable that is explicitly defined as a [ref local](#).

Here's a more complete ref return example, showing both the method signature and method body.

```
public static ref int Find(int[,] matrix, Func<int, bool> predicate)
{
    for (int i = 0; i < matrix.GetLength(0); i++)
        for (int j = 0; j < matrix.GetLength(1); j++)
            if (predicate(matrix[i, j]))
                return ref matrix[i, j];
    throw new InvalidOperationException("Not found");
}
```

The called method may also declare the return value as `ref readonly` to return the value by reference, and enforce that the calling code can't modify the returned value. The calling method can avoid copying the returned value by storing the value in a local [ref readonly](#) variable.

For an example, see [A ref returns and ref locals example](#).

Ref locals

A ref local variable is used to refer to values returned using `return ref`. A ref local variable cannot be initialized to a non-ref return value. In other words, the right-hand side of the initialization must be a reference. Any modifications to the value of the ref local are reflected in the state of the object whose method returned the value by reference.

You define a ref local by using the `ref` keyword in two places:

- Before the variable declaration.
- Immediately before the call to the method that returns the value by reference.

For example, the following statement defines a ref local value that is returned by a method named `GetEstimatedValue`:

```
ref decimal estValue = ref Building.GetEstimatedValue();
```

You can access a value by reference in the same way. In some cases, accessing a value by reference increases performance by avoiding a potentially expensive copy operation. For example, the following statement shows how to define a ref local variable that is used to reference a value.

```
ref VeryLargeStruct reflocal = ref veryLargeStruct;
```

In both examples the `ref` keyword must be used in both places, or the compiler generates error CS8172, "Cannot initialize a by-reference variable with a value."

Beginning with C# 7.3, the iteration variable of the `foreach` statement can be a ref local or ref readonly local variable. For more information, see the [foreach statement](#) article.

Also beginning with C# 7.3, you can reassign a ref local or ref readonly local variable with the [ref assignment operator](#).

Ref readonly locals

A ref readonly local is used to refer to values returned by a method or property that has `ref readonly` in its

signature and uses `return ref`. A `ref readonly` variable combines the properties of a `ref` local variable with a `readonly` variable: it's an alias to the storage it's assigned to, and it cannot be modified.

A ref returns and ref locals example

The following example defines a `Book` class that has two `String` fields, `Title` and `Author`. It also defines a `BookCollection` class that includes a private array of `Book` objects. Individual book objects are returned by reference by calling its `GetBookByTitle` method.

```
public class Book
{
    public string Author;
    public string Title;
}

public class BookCollection
{
    private Book[] books = { new Book { Title = "Call of the Wild, The", Author = "Jack London" },
                             new Book { Title = "Tale of Two Cities, A", Author = "Charles Dickens" }
    };
    private Book nobook = null;

    public ref Book GetBookByTitle(string title)
    {
        for (int ctr = 0; ctr < books.Length; ctr++)
        {
            if (title == books[ctr].Title)
                return ref books[ctr];
        }
        return ref nobook;
    }

    public void ListBooks()
    {
        foreach (var book in books)
        {
            Console.WriteLine($"{book.Title}, by {book.Author}");
        }
        Console.WriteLine();
    }
}
```

When the caller stores the value returned by the `GetBookByTitle` method as a ref local, changes that the caller makes to the return value are reflected in the `BookCollection` object, as the following example shows.

```
var bc = new BookCollection();
bc.ListBooks();

ref var book = ref bc.GetBookByTitle("Call of the Wild, The");
if (book != null)
    book = new Book { Title = "Republic, The", Author = "Plato" };
bc.ListBooks();
// The example displays the following output:
//      Call of the Wild, The, by Jack London
//      Tale of Two Cities, A, by Charles Dickens
//
//      Republic, The, by Plato
//      Tale of Two Cities, A, by Charles Dickens
```

For more information, see the [C# Language Specification](#). The language specification is the definitive source for C# syntax and usage.

See also

- [Write safe efficient code](#)
- [Ref returns and ref locals](#)
- [Conditional ref expression](#)
- [Passing Parameters](#)
- [Method Parameters](#)
- [C# Reference](#)
- [C# Programming Guide](#)
- [C# Keywords](#)

out parameter modifier (C# Reference)

12/28/2021 • 4 minutes to read • [Edit Online](#)

The `out` keyword causes arguments to be passed by reference. It makes the formal parameter an alias for the argument, which must be a variable. In other words, any operation on the parameter is made on the argument. It is like the `ref` keyword, except that `ref` requires that the variable be initialized before it is passed. It is also like the `in` keyword, except that `in` does not allow the called method to modify the argument value. To use an `out` parameter, both the method definition and the calling method must explicitly use the `out` keyword. For example:

```
int initializeInMethod;  
OutArgExample(out initializeInMethod);  
Console.WriteLine(initializeInMethod);    // value is now 44  
  
void OutArgExample(out int number)  
{  
    number = 44;  
}
```

NOTE

The `out` keyword can also be used with a generic type parameter to specify that the type parameter is covariant. For more information on the use of the `out` keyword in this context, see [out \(Generic Modifier\)](#).

Variables passed as `out` arguments do not have to be initialized before being passed in a method call. However, the called method is required to assign a value before the method returns.

The `in`, `ref`, and `out` keywords are not considered part of the method signature for the purpose of overload resolution. Therefore, methods cannot be overloaded if the only difference is that one method takes a `ref` or `in` argument and the other takes an `out` argument. The following code, for example, will not compile:

```
class CS0663_Example  
{  
    // Compiler error CS0663: "Cannot define overloaded  
    // methods that differ only on ref and out".  
    public void SampleMethod(out int i) { }  
    public void SampleMethod(ref int i) { }  
}
```

Overloading is legal, however, if one method takes a `ref`, `in`, or `out` argument and the other has none of those modifiers, like this:

```
class OutOverloadExample  
{  
    public void SampleMethod(int i) { }  
    public void SampleMethod(out int i) => i = 5;  
}
```

The compiler chooses the best overload by matching the parameter modifiers at the call site to the parameter modifiers used in the method call.

Properties are not variables and therefore cannot be passed as `out` parameters.

You can't use the `in`, `ref`, and `out` keywords for the following kinds of methods:

- Async methods, which you define by using the `async` modifier.
- Iterator methods, which include a `yield return` or `yield break` statement.

In addition, `extension methods` have the following restrictions:

- The `out` keyword cannot be used on the first argument of an extension method.
- The `ref` keyword cannot be used on the first argument of an extension method when the argument is not a struct, or a generic type not constrained to be a struct.
- The `in` keyword cannot be used unless the first argument is a struct. The `in` keyword cannot be used on any generic type, even when constrained to be a struct.

Declaring `out` parameters

Declaring a method with `out` arguments is a classic workaround to return multiple values. Beginning with C# 7.0, consider `value tuples` for similar scenarios. The following example uses `out` to return three variables with a single method call. The third argument is assigned to null. This enables methods to return values optionally.

```
void Method(out int answer, out string message, out string stillNull)
{
    answer = 44;
    message = "I've been returned";
    stillNull = null;
}

int argNumber;
string argMessage, argDefault;
Method(out argNumber, out argMessage, out argDefault);
Console.WriteLine(argNumber);
Console.WriteLine(argMessage);
Console.WriteLine(argDefault == null);

// The example displays the following output:
//      44
//      I've been returned
//      True
```

Calling a method with an `out` argument

In C# 6 and earlier, you must declare a variable in a separate statement before you pass it as an `out` argument. The following example declares a variable named `number` before it is passed to the `Int32.TryParse` method, which attempts to convert a string to a number.

```
string numberAsString = "1640";

int number;
if (Int32.TryParse(numberAsString, out number))
    Console.WriteLine($"Converted '{numberAsString}' to {number}");
else
    Console.WriteLine($"Unable to convert '{numberAsString}'");
// The example displays the following output:
//      Converted '1640' to 1640
```

Starting with C# 7.0, you can declare the `out` variable in the argument list of the method call, rather than in a separate variable declaration. This produces more compact, readable code, and also prevents you from inadvertently assigning a value to the variable before the method call. The following example is like the previous example, except that it defines the `number` variable in the call to the [Int32.TryParse](#) method.

```
string numberAsString = "1640";

if (Int32.TryParse(numberAsString, out int number))
    Console.WriteLine($"Converted '{numberAsString}' to {number}");
else
    Console.WriteLine($"Unable to convert '{numberAsString}'");
// The example displays the following output:
//      Converted '1640' to 1640
```

In the previous example, the `number` variable is strongly typed as an `int`. You can also declare an implicitly typed local variable, as the following example does.

```
string numberAsString = "1640";

if (Int32.TryParse(numberAsString, out var number))
    Console.WriteLine($"Converted '{numberAsString}' to {number}");
else
    Console.WriteLine($"Unable to convert '{numberAsString}'");
// The example displays the following output:
//      Converted '1640' to 1640
```

C# Language Specification

For more information, see the [C# Language Specification](#). The language specification is the definitive source for C# syntax and usage.

See also

- [C# Reference](#)
- [C# Programming Guide](#)
- [C# Keywords](#)
- [Method Parameters](#)

namespace

12/28/2021 • 2 minutes to read • [Edit Online](#)

The `namespace` keyword is used to declare a scope that contains a set of related objects. You can use a namespace to organize code elements and to create globally unique types.

```
namespace SampleNamespace
{
    class SampleClass { }

    interface ISampleInterface { }

    struct SampleStruct { }

    enum SampleEnum { a, b }

    delegate void SampleDelegate(int i);

    namespace Nested
    {
        class SampleClass2 { }
    }
}
```

File scoped namespace declarations enable you to declare that all types in a file are in a single namespace. File scoped namespace declarations are available with C# 10. The following example is similar to the previous example, but uses a file scoped namespace declaration:

```
using System;

namespace SampleFileScopedNamespace;

class SampleClass { }

interface ISampleInterface { }

struct SampleStruct { }

enum SampleEnum { a, b }

delegate void SampleDelegate(int i);
```

The preceding example doesn't include a nested namespace. File scoped namespaces can't include additional namespace declarations. You cannot declare a nested namespace or a second file-scoped namespace:

```

namespace SampleNamespace;

class AnotherSampleClass
{
    public void AnotherSampleMethod()
    {
        System.Console.WriteLine(
            "SampleMethod inside SampleNamespace");
    }
}

namespace AnotherNamespace; // Not allowed!

namespace ANestedNamespace // Not allowed!
{
    // declarations...
}

```

Within a namespace, you can declare zero or more of the following types:

- [class](#)
- [interface](#)
- [struct](#)
- [enum](#)
- [delegate](#)
- nested namespaces can be declared except in file scoped namespace declarations

The compiler adds a default namespace. This unnamed namespace, sometimes referred to as the global namespace, is present in every file. It contains declarations not included in a declared namespace. Any identifier in the global namespace is available for use in a named namespace.

Namespaces implicitly have public access. For a discussion of the access modifiers you can assign to elements in a namespace, see [Access Modifiers](#).

It's possible to define a namespace in two or more declarations. For example, the following example defines two classes as part of the `MyCompany` namespace:

```

namespace MyCompany.Proj1
{
    class MyClass
    {
    }
}

namespace MyCompany.Proj1
{
    class MyClass1
    {
    }
}

```

The following example shows how to call a static method in a nested namespace.

```
namespace SomeNameSpace
{
    public class MyClass
    {
        static void Main()
        {
            Nested.NestedNameSpaceClass.SayHello();
        }
    }

    // a nested namespace
    namespace Nested
    {
        public class NestedNameSpaceClass
        {
            public static void SayHello()
            {
                Console.WriteLine("Hello");
            }
        }
    }
}
// Output: Hello
```

C# language specification

For more information, see the [Namespaces](#) section of the [C# language specification](#). For more information on file scoped namespace declarations, see the [feature specification](#).

See also

- [C# reference](#)
- [C# keywords](#)
- [using](#)
- [using static](#)
- [Namespace alias qualifier](#) `::`
- [Namespaces](#)

using (C# Reference)

12/28/2021 • 2 minutes to read • [Edit Online](#)

The `using` keyword has two major uses:

- The [using statement](#) defines a scope at the end of which an object will be disposed.
- The [using directive](#) creates an alias for a namespace or imports types defined in other namespaces.

See also

- [C# Reference](#)
- [C# Programming Guide](#)
- [C# Keywords](#)
- [Namespaces](#)
- [extern](#)

using directive

12/28/2021 • 9 minutes to read • [Edit Online](#)

The `using` directive allows you to use types defined in a namespace without specifying the fully qualified namespace of that type. In its basic form, the `using` directive imports all the types from a single namespace, as shown in the following example:

```
using System.Text;
```

You can apply two modifiers to a `using` directive:

- The `global` modifier has the same effect as adding the same `using` directive to every source file in your project. This modifier was introduced in C# 10.
- The `static` modifier imports the `static` members and nested types from a single type rather than importing all the types in a namespace. This modifier was introduced in C# 6.0.

You can combine both modifiers to import the static members from a type in all source files in your project.

You can also create an alias for a namespace or a type with a *using alias directive*.

```
using Project = PC.MyCompany.Project;
```

You can use the `global` modifier on a *using alias directive*.

NOTE

The `using` keyword is also used to create *using statements*, which help ensure that `IDisposable` objects such as files and fonts are handled correctly. For more information about the *using statement*, see [using Statement](#).

The scope of a `using` directive without the `global` modifier is the file in which it appears.

The `using` directive can appear:

- At the beginning of a source code file, before any namespace or type declarations.
- In any namespace, but before any namespaces or types declared in that namespace, unless the `global` modifier is used, in which case the directive must appear before all namespace and type declarations.

Otherwise, compiler error [CS1529](#) is generated.

Create a `using` directive to use the types in a namespace without having to specify the namespace. A `using` directive doesn't give you access to any namespaces that are nested in the namespace you specify. Namespaces come in two categories: user-defined and system-defined. User-defined namespaces are namespaces defined in your code. For a list of the system-defined namespaces, see [.NET API Browser](#).

global modifier

Adding the `global` modifier to a `using` directive means that using is applied to all files in the compilation (typically a project). The `global using` directive was added in C# 10. Its syntax is:


```
global using <fully-qualified-namespace>;
```

where *fully-qualified-namespace* is the fully qualified name of the namespace whose types can be referenced without specifying the namespace.

A *global using* directive can appear at the beginning of any source code file. All `global using` directives in a single file must appear before:

- All `using` directives without the `global` modifier.
- All namespace and type declarations in the file.

You may add `global using` directives to any source file. Typically, you'll want to keep them in a single location. The order of `global using` directives doesn't matter, either in a single file, or between files.

The `global` modifier may be combined with the `static` modifier. The `global` modifier may be applied to a *using alias directive*. In both cases, the directive's scope is all files in the current compilation. The following example enables using all the methods declared in the [System.Math](#) in all files in your project:

```
global using static System.Math;
```

You can also globally include a namespace by adding a `<Using>` item to your project file, for example, `<Using Include="My.Awesome.Namespace" />`. For more information, see [<Using> item](#).

IMPORTANT

The C# templates for .NET 6 use *top level statements*. Your application may not match the code in this article, if you've already upgraded to the .NET 6 previews. For more information see the article on [New C# templates generate top level statements](#)

The .NET 6 SDK also adds a set of *implicit* `global using` directives for projects that use the following SDKs:

- Microsoft.NET.Sdk
- Microsoft.NET.Sdk.Web
- Microsoft.NET.Sdk.Worker

These implicit `global using` directives include the most common namespaces for the project type.

static modifier

The `using static` directive names a type whose static members and nested types you can access without specifying a type name. The `using static` directive was introduced in C# 6. Its syntax is:

```
using static <fully-qualified-type-name>;
```

The `<fully-qualified-type-name>` is the name of the type whose static members and nested types can be referenced without specifying a type name. If you don't provide a fully qualified type name (the full namespace name along with the type name), C# generates compiler error [CS0246](#): "The type or namespace name 'type/namespace' couldn't be found (are you missing a using directive or an assembly reference?)".

The `using static` directive applies to any type that has static members (or nested types), even if it also has instance members. However, instance members can only be invoked through the type instance.

You can access static members of a type without having to qualify the access with the type name:

```
using static System.Console;
using static System.Math;
class Program
{
    static void Main()
    {
        WriteLine(Sqrt(3*3 + 4*4));
    }
}
```

Ordinarily, when you call a static member, you provide the type name along with the member name. Repeatedly entering the same type name to invoke members of the type can result in verbose, obscure code. For example, the following definition of a `Circle` class references many members of the `Math` class.

```
using System;

public class Circle
{
    public Circle(double radius)
    {
        Radius = radius;
    }

    public double Radius { get; set; }

    public double Diameter
    {
        get { return 2 * Radius; }
    }

    public double Circumference
    {
        get { return 2 * Radius * Math.PI; }
    }

    public double Area
    {
        get { return Math.PI * Math.Pow(Radius, 2); }
    }
}
```

By eliminating the need to explicitly reference the `Math` class each time a member is referenced, the `using static` directive produces cleaner code:

```

using System;
using static System.Math;

public class Circle
{
    public Circle(double radius)
    {
        Radius = radius;
    }

    public double Radius { get; set; }

    public double Diameter
    {
        get { return 2 * Radius; }
    }

    public double Circumference
    {
        get { return 2 * Radius * PI; }
    }

    public double Area
    {
        get { return PI * Pow(Radius, 2); }
    }
}

```

`using static` imports only accessible static members and nested types declared in the specified type. Inherited members aren't imported. You can import from any named type with a `using static` directive, including Visual Basic modules. If F# top-level functions appear in metadata as static members of a named type whose name is a valid C# identifier, then the F# functions can be imported.

`using static` makes extension methods declared in the specified type available for extension method lookup. However, the names of the extension methods aren't imported into scope for unqualified reference in code.

Methods with the same name imported from different types by different `using static` directives in the same compilation unit or namespace form a method group. Overload resolution within these method groups follows normal C# rules.

The following example uses the `using static` directive to make the static members of the [Console](#), [Math](#), and [String](#) classes available without having to specify their type name.

```

using System;
using static System.Console;
using static System.Math;
using static System.String;

class Program
{
    static void Main()
    {
        Write("Enter a circle's radius: ");
        var input = ReadLine();
        if (!IsNullOrEmpty(input) && double.TryParse(input, out var radius)) {
            var c = new Circle(radius);

            string s = "\nInformation about the circle:\n";
            s = s + Format("    Radius: {0:N2}\n", c.Radius);
            s = s + Format("    Diameter: {0:N2}\n", c.Diameter);
            s = s + Format("    Circumference: {0:N2}\n", c.Circumference);
            s = s + Format("    Area: {0:N2}\n", c.Area);
            WriteLine(s);
        }
        else {
            WriteLine("Invalid input...");
        }
    }
}

public class Circle
{
    public Circle(double radius)
    {
        Radius = radius;
    }

    public double Radius { get; set; }

    public double Diameter
    {
        get { return 2 * Radius; }
    }

    public double Circumference
    {
        get { return 2 * Radius * PI; }
    }

    public double Area
    {
        get { return PI * Pow(Radius, 2); }
    }
}

// The example displays the following output:
//      Enter a circle's radius: 12.45
//
//      Information about the circle:
//          Radius: 12.45
//          Diameter: 24.90
//          Circumference: 78.23
//          Area: 486.95

```

In the example, the `using static` directive could also have been applied to the `Double` type. Adding that directive would make it possible to call the `TryParse(String, Double)` method without specifying a type name. However, using `TryParse` without a type name creates less readable code, since it becomes necessary to check the `using static` directives to determine which numeric type's `TryParse` method is called.

`using static` also applies to `enum` types. By adding `using static` with the enum, the type is no longer required to use the enum members.

```
using static Color;

enum Color
{
    Red,
    Green,
    Blue
}

class Program
{
    public static void Main()
    {
        Color color = Green;
    }
}
```

using alias

Create a `using` alias directive to make it easier to qualify an identifier to a namespace or type. In any `using` directive, the fully qualified namespace or type must be used regardless of the `using` directives that come before it. No `using` alias can be used in the declaration of a `using` directive. For example, the following example generates a compiler error:

```
using s = System.Text;
using s.RegularExpressions; // Generates a compiler error.
```

The following example shows how to define and use a `using` alias for a namespace:

```
namespace PC
{
    // Define an alias for the nested namespace.
    using Project = PC.MyCompany.Project;
    class A
    {
        void M()
        {
            // Use the alias
            var mc = new Project.MyClass();
        }
    }
    namespace MyCompany
    {
        namespace Project
        {
            public class MyClass { }
        }
    }
}
```

A `using` alias directive can't have an open generic type on the right-hand side. For example, you can't create a `using` alias for a `List<T>`, but you can create one for a `List<int>`.

The following example shows how to define a `using` directive and a `using` alias for a class:

```

using System;

// Using alias directive for a class.
using AliasToMyClass = NameSpace1.MyClass;

// Using alias directive for a generic class.
using UsingAlias = NameSpace2.MyClass<int>;

namespace NameSpace1
{
    public class MyClass
    {
        public override string ToString()
        {
            return "You are in NameSpace1.MyClass.";
        }
    }
}

namespace NameSpace2
{
    class MyClass<T>
    {
        public override string ToString()
        {
            return "You are in NameSpace2.MyClass.";
        }
    }
}

namespace NameSpace3
{
    class MainClass
    {
        static void Main()
        {
            var instance1 = new AliasToMyClass();
            Console.WriteLine(instance1);

            var instance2 = new UsingAlias();
            Console.WriteLine(instance2);
        }
    }
}
// Output:
//   You are in NameSpace1.MyClass.
//   You are in NameSpace2.MyClass.

```

How to use the Visual Basic `My` namespace

The [Microsoft.VisualBasic.MyServices](#) namespace (`My` in Visual Basic) provides easy and intuitive access to a number of .NET classes, enabling you to write code that interacts with the computer, application, settings, resources, and so on. Although originally designed for use with Visual Basic, the `MyServices` namespace can be used in C# applications.

For more information about using the `MyServices` namespace from Visual Basic, see [Development with My](#).

You need to add a reference to the *Microsoft.VisualBasic.dll* assembly in your project. Not all the classes in the `MyServices` namespace can be called from a C# application: for example, the [FileSystemProxy](#) class is not compatible. In this particular case, the static methods that are part of [FileSystem](#), which are also contained in *VisualBasic.dll*, can be used instead. For example, here is how to use one such method to duplicate a directory:

```
// Duplicate a directory
Microsoft.VisualBasic.FileIO.FileSystem.CopyDirectory(
    @"C:\original_directory",
    @"C:\copy_of_original_directory");
```

C# language specification

For more information, see [Using directives](#) in the [C# Language Specification](#). The language specification is the definitive source for C# syntax and usage.

For more information on the *global using* modifier, see the [global usings feature specification - C# 10](#).

See also

- [C# Reference](#)
- [C# Programming Guide](#)
- [C# Keywords](#)
- [Namespaces](#)
- [using Statement](#)

using statement (C# Reference)

12/28/2021 • 4 minutes to read • [Edit Online](#)

Provides a convenient syntax that ensures the correct use of [IDisposable](#) objects. Beginning in C# 8.0, the `using` statement ensures the correct use of [IAsyncDisposable](#) objects.

Example

The following example shows how to use the `using` statement.

```
string manyLines = @"This is line one
This is line two
Here is line three
The penultimate line is line four
This is the final, fifth line.";

using (var reader = new StringReader(manyLines))
{
    string? item;
    do
    {
        item = reader.ReadLine();
        Console.WriteLine(item);
    } while (item != null);
}
```

Beginning with C# 8.0, you can use the following alternative syntax for the `using` statement that doesn't require braces:

```
string manyLines = @"This is line one
This is line two
Here is line three
The penultimate line is line four
This is the final, fifth line.";

using var reader = new StringReader(manyLines);
string? item;
do
{
    item = reader.ReadLine();
    Console.WriteLine(item);
} while (item != null);
```

Remarks

[File](#) and [Font](#) are examples of managed types that access unmanaged resources (in this case file handles and device contexts). There are many other kinds of unmanaged resources and class library types that encapsulate them. All such types must implement the [IDisposable](#) interface, or the [IAsyncDisposable](#) interface.

When the lifetime of an `IDisposable` object is limited to a single method, you should declare and instantiate it in the `using` statement. The `using` statement calls the [Dispose](#) method on the object in the correct way, and (when you use it as shown earlier) it also causes the object itself to go out of scope as soon as [Dispose](#) is called. Within the `using` block, the object is read-only and can't be modified or reassigned. If the object implements `IAsyncDisposable` instead of `IDisposable`, the `using` statement calls the [DisposeAsync](#) and `await`s the

returned [ValueTask](#). For more information on [IAsyncDisposable](#), see [Implement a DisposeAsync method](#).

The `using` statement ensures that [Dispose](#) (or [DisposeAsync](#)) is called even if an exception occurs within the `using` block. You can achieve the same result by putting the object inside a `try` block and then calling [Dispose](#) (or [DisposeAsync](#)) in a `finally` block; in fact, this is how the `using` statement is translated by the compiler. The code example earlier expands to the following code at compile time (note the extra curly braces to create the limited scope for the object):

```
string manyLines = @"This is line one
This is line two
Here is line three
The penultimate line is line four
This is the final, fifth line.";

{
    var reader = new StringReader(manyLines);
    try
    {
        string? item;
        do
        {
            item = reader.ReadLine();
            Console.WriteLine(item);
        } while (item != null);
    }
    finally
    {
        reader?.Dispose();
    }
}
```

The newer `using` statement syntax translates to similar code. The `try` block opens where the variable is declared. The `finally` block is added at the close of the enclosing block, typically at the end of a method.

For more information about the `try` - `finally` statement, see the [try-finally](#) article.

Multiple instances of a type can be declared in a single `using` statement, as shown in the following example. Notice that you can't use implicitly typed variables (`var`) when you declare multiple variables in a single statement:

```
string numbers = @"One
Two
Three
Four.";
string letters = @"A
B
C
D.";

using (StringReader left = new StringReader(numbers),
    right = new StringReader(letters))
{
    string? item;
    do
    {
        item = left.ReadLine();
        Console.Write(item);
        Console.Write(" ");
        item = right.ReadLine();
        Console.WriteLine(item);
    } while (item != null);
}
```

You can combine multiple declarations of the same type using the new syntax introduced with C# 8 as well, as shown in the following example:

```
string numbers = @"One
Two
Three
Four.";
string letters = @"A
B
C
D.";

using StringReader left = new StringReader(numbers),
    right = new StringReader(letters);
string? item;
do
{
    item = left.ReadLine();
    Console.Write(item);
    Console.Write(" ");
    item = right.ReadLine();
    Console.WriteLine(item);
} while (item != null);
```

You can instantiate the resource object and then pass the variable to the `using` statement, but this isn't a best practice. In this case, after control leaves the `using` block, the object remains in scope but probably has no access to its unmanaged resources. In other words, it's not fully initialized anymore. If you try to use the object outside the `using` block, you risk causing an exception to be thrown. For this reason, it's better to instantiate the object in the `using` statement and limit its scope to the `using` block.

```
string manyLines = @"This is line one
This is line two
Here is line three
The penultimate line is line four
This is the final, fifth line.";

var reader = new StringReader(manyLines);
using (reader)
{
    string? item;
    do
    {
        item = reader.ReadLine();
        Console.WriteLine(item);
    } while (item != null);
}
// reader is in scope here, but has been disposed
```

For more information about disposing of `IDisposable` objects, see [Using objects that implement IDisposable](#).

C# language specification

For more information, see [The using statement](#) in the [C# Language Specification](#). The language specification is the definitive source for C# syntax and usage.

See also

- [C# Reference](#)
- [C# Programming Guide](#)

- C# Keywords
- using Directive
- Garbage Collection
- Using objects that implement IDisposable
- IDisposable interface
- using statement in C# 8.0

extern alias (C# Reference)

12/28/2021 • 2 minutes to read • [Edit Online](#)

You might have to reference two versions of assemblies that have the same fully-qualified type names. For example, you might have to use two or more versions of an assembly in the same application. By using an external assembly alias, the namespaces from each assembly can be wrapped inside root-level namespaces named by the alias, which enables them to be used in the same file.

NOTE

The `extern` keyword is also used as a method modifier, declaring a method written in unmanaged code.

To reference two assemblies with the same fully-qualified type names, an alias must be specified at a command prompt, as follows:

```
/r:GridV1=grid.dll
```

```
/r:GridV2=grid20.dll
```

This creates the external aliases `GridV1` and `GridV2`. To use these aliases from within a program, reference them by using the `extern` keyword. For example:

```
extern alias GridV1;
```

```
extern alias GridV2;
```

Each extern alias declaration introduces an additional root-level namespace that parallels (but does not lie within) the global namespace. Thus types from each assembly can be referred to without ambiguity by using their fully qualified name, rooted in the appropriate namespace-alias.

In the previous example, `GridV1::Grid` would be the grid control from `grid.dll`, and `GridV2::Grid` would be the grid control from `grid20.dll`.

Using Visual Studio

If you are using Visual Studio, aliases can be provided in similar way.

Add reference of `grid.dll` and `grid20.dll` to your project in Visual Studio. Open a property tab and change the Aliases from global to GridV1 and GridV2 respectively.

Use these aliases the same way above

```
extern alias GridV1;
```

```
extern alias GridV2;
```

Now you can create alias for a namespace or a type by *using alias directive*. For more information, see [using directive](#).

```
using Class1V1 = GridV1::Namespace.Class1;
```

```
using Class1V2 = GridV2::Namespace.Class1;
```

C# Language Specification

For more information, see the [C# Language Specification](#). The language specification is the definitive source for C# syntax and usage.

See also

- [C# Reference](#)
- [C# Programming Guide](#)
- [C# Keywords](#)
- [:: Operator](#)
- [References \(C# Compiler Options\)](#)

new constraint (C# Reference)

12/28/2021 • 2 minutes to read • [Edit Online](#)

The `new` constraint specifies that a type argument in a generic class declaration must have a public parameterless constructor. To use the `new` constraint, the type cannot be abstract.

Apply the `new` constraint to a type parameter when a generic class creates new instances of the type, as shown in the following example:

```
class ItemFactory<T> where T : new()
{
    public T GetNewItem()
    {
        return new T();
    }
}
```

When you use the `new()` constraint with other constraints, it must be specified last:

```
public class ItemFactory2<T>
    where T : IComparable, new()
{ }
```

For more information, see [Constraints on Type Parameters](#).

You can also use the `new` keyword to [create an instance of a type](#) or as a [member declaration modifier](#).

C# language specification

For more information, see the [Type parameter constraints](#) section of the [C# language specification](#).

See also

- [C# Reference](#)
- [C# Programming Guide](#)
- [C# Keywords](#)
- [Generics](#)

where (generic type constraint) (C# Reference)

12/28/2021 • 5 minutes to read • [Edit Online](#)

The `where` clause in a generic definition specifies constraints on the types that are used as arguments for type parameters in a generic type, method, delegate, or local function. Constraints can specify interfaces, base classes, or require a generic type to be a reference, value, or unmanaged type. They declare capabilities that the type argument must have.

For example, you can declare a generic class, `AGenericClass`, such that the type parameter `T` implements the `IComparable<T>` interface:

```
public class AGenericClass<T> where T : IComparable<T> { }
```

NOTE

For more information on the `where` clause in a query expression, see [where clause](#).

The `where` clause can also include a base class constraint. The base class constraint states that a type to be used as a type argument for that generic type has the specified class as a base class, or is that base class. If the base class constraint is used, it must appear before any other constraints on that type parameter. Some types are disallowed as a base class constraint: `Object`, `Array`, and `ValueType`. Before C# 7.3, `Enum`, `Delegate`, and `MulticastDelegate` were also disallowed as base class constraints. The following example shows the types that can now be specified as a base class:

```
public class UsingEnum<T> where T : System.Enum { }

public class UsingDelegate<T> where T : System.Delegate { }

public class Multicaster<T> where T : System.MulticastDelegate { }
```

In a nullable context in C# 8.0 and later, the nullability of the base class type is enforced. If the base class is non-nullable (for example `Base`), the type argument must be non-nullable. If the base class is nullable (for example `Base?`), the type argument may be either a nullable or non-nullable reference type. The compiler issues a warning if the type argument is a nullable reference type when the base class is non-nullable.

The `where` clause can specify that the type is a `class` or a `struct`. The `struct` constraint removes the need to specify a base class constraint of `System.ValueType`. The `System.ValueType` type may not be used as a base class constraint. The following example shows both the `class` and `struct` constraints:

```
class MyClass<T, U>
    where T : class
    where U : struct
{ }
```

In a nullable context in C# 8.0 and later, the `class` constraint requires a type to be a non-nullable reference type. To allow nullable reference types, use the `class?` constraint, which allows both nullable and non-nullable reference types.

The `where` clause may include the `notnull` constraint. The `notnull` constraint limits the type parameter to

non-nullable types. The type may be a [value type](#) or a non-nullable reference type. The `notnull` constraint is available starting in C# 8.0 for code compiled in a `nullable enable` context. Unlike other constraints, if a type argument violates the `notnull` constraint, the compiler generates a warning instead of an error. Warnings are only generated in a `nullable enable` context.

The addition of nullable reference types introduces a potential ambiguity in the meaning of `T?` in generic methods. If `T` is a `struct`, `T?` is the same as `System.Nullable<T>`. However, if `T` is a reference type, `T?` means that `null` is a valid value. The ambiguity arises because overriding methods can't include constraints. The new `default` constraint resolves this ambiguity. You'll add it when a base class or interface declares two overloads of a method, one that specifies the `struct` constraint, and one that doesn't have either the `struct` or `class` constraint applied:

```
public abstract class B
{
    public void M<T>(T? item) where T : struct { }
    public abstract void M<T>(T? item);
}
```

You use the `default` constraint to specify that your derived class overrides the method without the constraint in your derived class, or explicit interface implementation. It's only valid on methods that override base methods, or explicit interface implementations:

```
public class D : B
{
    // Without the "default" constraint, the compiler tries to override the first method in B
    public override void M<T>(T? item) where T : default { }
```

IMPORTANT

Generic declarations that include the `notnull` constraint can be used in a nullable oblivious context, but compiler does not enforce the constraint.

```
#nullable enable
class NotNullContainer<T>
    where T : notnull
{
}
#nullable restore
```

The `where` clause may also include an `unmanaged` constraint. The `unmanaged` constraint limits the type parameter to types known as [unmanaged types](#). The `unmanaged` constraint makes it easier to write low-level interop code in C#. This constraint enables reusable routines across all unmanaged types. The `unmanaged` constraint can't be combined with the `class` or `struct` constraint. The `unmanaged` constraint enforces that the type must be a `struct`:

```
class UnManagedWrapper<T>
    where T : unmanaged
{ }
```

The `where` clause may also include a constructor constraint, `new()`. That constraint makes it possible to create an instance of a type parameter using the `new` operator. The [new\(\) Constraint](#) lets the compiler know that any

type argument supplied must have an accessible parameterless constructor. For example:

```
public class MyGenericClass<T> where T : IComparable<T>, new()
{
    // The following line is not possible without new() constraint:
    T item = new T();
}
```

The `new()` constraint appears last in the `where` clause. The `new()` constraint can't be combined with the `struct` or `unmanaged` constraints. All types satisfying those constraints must have an accessible parameterless constructor, making the `new()` constraint redundant.

With multiple type parameters, use one `where` clause for each type parameter, for example:

```
public interface IMyInterface { }

namespace CodeExample
{
    class Dictionary<TKey, TVal>
        where TKey : IComparable<TKey>
        where TVal : IMyInterface
    {
        public void Add(TKey key, TVal val) { }
    }
}
```

You can also attach constraints to type parameters of generic methods, as shown in the following example:

```
public void MyMethod<T>(T t) where T : IMyInterface { }
```

Notice that the syntax to describe type parameter constraints on delegates is the same as that of methods:

```
delegate T MyDelegate<T>() where T : new();
```

For information on generic delegates, see [Generic Delegates](#).

For details on the syntax and use of constraints, see [Constraints on Type Parameters](#).

C# language specification

For more information, see the [C# Language Specification](#). The language specification is the definitive source for C# syntax and usage.

See also

- [C# Reference](#)
- [C# Programming Guide](#)
- [Introduction to Generics](#)
- [new Constraint](#)
- [Constraints on Type Parameters](#)

base (C# Reference)

12/28/2021 • 2 minutes to read • [Edit Online](#)

The `base` keyword is used to access members of the base class from within a derived class:

- Call a method on the base class that has been overridden by another method.
- Specify which base-class constructor should be called when creating instances of the derived class.

A base class access is permitted only in a constructor, an instance method, or an instance property accessor.

It is an error to use the `base` keyword from within a static method.

The base class that is accessed is the base class specified in the class declaration. For example, if you specify `class ClassB : ClassA`, the members of ClassA are accessed from ClassB, regardless of the base class of ClassA.

Example 1

In this example, both the base class, `Person`, and the derived class, `Employee`, have a method named `Getinfo`. By using the `base` keyword, it is possible to call the `Getinfo` method on the base class, from within the derived class.

```

public class Person
{
    protected string ssn = "444-55-6666";
    protected string name = "John L. Malgraine";

    public virtual void GetInfo()
    {
        Console.WriteLine("Name: {0}", name);
        Console.WriteLine("SSN: {0}", ssn);
    }
}

class Employee : Person
{
    public string id = "ABC567EFG";
    public override void GetInfo()
    {
        // Calling the base class GetInfo method:
        base.GetInfo();
        Console.WriteLine("Employee ID: {0}", id);
    }
}

class TestClass
{
    static void Main()
    {
        Employee E = new Employee();
        E.GetInfo();
    }
}
/*
Output
Name: John L. Malgraine
SSN: 444-55-6666
Employee ID: ABC567EFG
*/

```

For additional examples, see [new](#), [virtual](#), and [override](#).

Example 2

This example shows how to specify the base-class constructor called when creating instances of a derived class.

```

public class BaseClass
{
    int num;

    public BaseClass()
    {
        Console.WriteLine("in BaseClass()");
    }

    public BaseClass(int i)
    {
        num = i;
        Console.WriteLine("in BaseClass(int i)");
    }

    public int GetNum()
    {
        return num;
    }
}

public class DerivedClass : BaseClass
{
    // This constructor will call BaseClass.BaseClass()
    public DerivedClass() : base()
    {
    }

    // This constructor will call BaseClass.BaseClass(int i)
    public DerivedClass(int i) : base(i)
    {
    }

    static void Main()
    {
        DerivedClass md = new DerivedClass();
        DerivedClass md1 = new DerivedClass(1);
    }
}
/*
Output:
in BaseClass()
in BaseClass(int i)
*/

```

C# language specification

For more information, see the [C# Language Specification](#). The language specification is the definitive source for C# syntax and usage.

See also

- [C# Reference](#)
- [C# Programming Guide](#)
- [C# Keywords](#)
- [this](#)

this (C# Reference)

12/28/2021 • 2 minutes to read • [Edit Online](#)

The `this` keyword refers to the current instance of the class and is also used as a modifier of the first parameter of an extension method.

NOTE

This article discusses the use of `this` with class instances. For more information about its use in extension methods, see [Extension Methods](#).

The following are common uses of `this`:

- To qualify members hidden by similar names, for example:

```
public class Employee
{
    private string alias;
    private string name;

    public Employee(string name, string alias)
    {
        // Use this to qualify the members of the class
        // instead of the constructor parameters.
        this.name = name;
        this.alias = alias;
    }
}
```

- To pass an object as a parameter to other methods, for example:

```
CalcTax(this);
```

- To declare indexers, for example:

```
public int this[int param]
{
    get { return array[param]; }
    set { array[param] = value; }
}
```

Static member functions, because they exist at the class level and not as part of an object, do not have a `this` pointer. It is an error to refer to `this` in a static method.

Example

In this example, `this` is used to qualify the `Employee` class members, `name` and `alias`, which are hidden by similar names. It is also used to pass an object to the method `CalcTax`, which belongs to another class.

```

class Employee
{
    private string name;
    private string alias;
    private decimal salary = 3000.00m;

    // Constructor:
    public Employee(string name, string alias)
    {
        // Use this to qualify the fields, name and alias:
        this.name = name;
        this.alias = alias;
    }

    // Printing method:
    public void printEmployee()
    {
        Console.WriteLine("Name: {0}\nAlias: {1}", name, alias);
        // Passing the object to the CalcTax method by using this:
        Console.WriteLine("Taxes: {0:C}", Tax.CalcTax(this));
    }

    public decimal Salary
    {
        get { return salary; }
    }
}

class Tax
{
    public static decimal CalcTax(Employee E)
    {
        return 0.08m * E.Salary;
    }
}

class MainClass
{
    static void Main()
    {
        // Create objects:
        Employee E1 = new Employee("Mingda Pan", "mpan");

        // Display results:
        E1.printEmployee();
    }
}
/*
Output:
    Name: Mingda Pan
    Alias: mpan
    Taxes: $240.00
*/

```

C# language specification

For more information, see the [C# Language Specification](#). The language specification is the definitive source for C# syntax and usage.

See also

- [C# Reference](#)
- [C# Programming Guide](#)

- [C# Keywords](#)
- [base](#)
- [Methods](#)

null (C# Reference)

12/28/2021 • 2 minutes to read • [Edit Online](#)

The `null` keyword is a literal that represents a null reference, one that does not refer to any object. `null` is the default value of reference-type variables. Ordinary value types cannot be null, except for [nullable value types](#).

The following example demonstrates some behaviors of the `null` keyword:


```

class Program
{
    class MyClass
    {
        public void MyMethod() { }
    }

    static void Main(string[] args)
    {
        // Set a breakpoint here to see that mc = null.
        // However, the compiler considers it "unassigned."
        // and generates a compiler error if you try to
        // use the variable.
        MyClass mc;

        // Now the variable can be used, but...
        mc = null;

        // ... a method call on a null object raises
        // a run-time NullReferenceException.
        // Uncomment the following line to see for yourself.
        // mc.MyMethod();

        // Now mc has a value.
        mc = new MyClass();

        // You can call its method.
        mc.MyMethod();

        // Set mc to null again. The object it referenced
        // is no longer accessible and can now be garbage-collected.
        mc = null;

        // A null string is not the same as an empty string.
        string s = null;
        string t = String.Empty; // Logically the same as ""

        // Equals applied to any null object returns false.
        bool b = (t.Equals(s));
        Console.WriteLine(b);

        // Equality operator also returns false when one
        // operand is null.
        Console.WriteLine("Empty string {0} null string", s == t ? "equals": "does not equal");

        // Returns true.
        Console.WriteLine("null == null is {0}", null == null);

        // A value type cannot be null
        // int i = null; // Compiler error!

        // Use a nullable value type instead:
        int? i = null;

        // Keep the console window open in debug mode.
        System.Console.WriteLine("Press any key to exit.");
        System.Console.ReadKey();
    }
}

```

C# language specification

For more information, see the [C# Language Specification](#). The language specification is the definitive source for C# syntax and usage.

See also

- [C# reference](#)
- [C# keywords](#)
- [Default values of C# types](#)
- [Nothing \(Visual Basic\)](#)

bool (C# reference)

12/28/2021 • 2 minutes to read • [Edit Online](#)

The `bool` type keyword is an alias for the .NET [System.Boolean](#) structure type that represents a Boolean value, which can be either `true` or `false`.

To perform logical operations with values of the `bool` type, use [Boolean logical](#) operators. The `bool` type is the result type of [comparison](#) and [equality](#) operators. A `bool` expression can be a controlling conditional expression in the [if](#), [do](#), [while](#), and [for](#) statements and in the [conditional operator](#) `?:`.

The default value of the `bool` type is `false`.

Literals

You can use the `true` and `false` literals to initialize a `bool` variable or to pass a `bool` value:

```
bool check = true;
Console.WriteLine(check ? "Checked" : "Not checked"); // output: Checked

Console.WriteLine(false ? "Checked" : "Not checked"); // output: Not checked
```

Three-valued Boolean logic

Use the nullable `bool?` type, if you need to support the three-valued logic, for example, when you work with databases that support a three-valued Boolean type. For the `bool?` operands, the predefined `&` and `|` operators support the three-valued logic. For more information, see the [Nullable Boolean logical operators](#) section of the [Boolean logical operators](#) article.

For more information about nullable value types, see [Nullable value types](#).

Conversions

C# provides only two conversions that involve the `bool` type. Those are an implicit conversion to the corresponding nullable `bool?` type and an explicit conversion from the `bool?` type. However, .NET provides additional methods that you can use to convert to or from the `bool` type. For more information, see the [Converting to and from Boolean values](#) section of the [System.Boolean](#) API reference page.

C# language specification

For more information, see [The bool type](#) section of the [C# language specification](#).

See also

- [C# reference](#)
- [Value types](#)
- [true and false operators](#)

default (C# reference)

12/28/2021 • 2 minutes to read • [Edit Online](#)

You can use the `default` keyword in the following contexts:

- To specify the default case in the `switch` statement.
- As the [default operator or literal](#) to produce the default value of a type.
- As the `default` [type constraint](#) on a generic method override or explicit interface implementation.

See also

- [C# reference](#)
- [C# keywords](#)

Contextual keywords (C# Reference)

12/28/2021 • 2 minutes to read • [Edit Online](#)

A contextual keyword is used to provide a specific meaning in the code, but it is not a reserved word in C#. The following contextual keywords are introduced in this section:

KEYWORD	DESCRIPTION
<code>add</code>	Defines a custom event accessor that is invoked when client code subscribes to the event.
<code>and</code>	Creates a pattern that matches when both of nested patterns match.
<code>async</code>	Indicates that the modified method, lambda expression, or anonymous method is asynchronous.
<code>await</code>	Suspends an async method until an awaited task is completed.
<code>dynamic</code>	Defines a reference type that enables operations in which it occurs to bypass compile-time type checking.
<code>get</code>	Defines an accessor method for a property or an indexer.
<code>global</code>	Alias of the global namespace, which is otherwise unnamed.
<code>init</code>	Defines an accessor method for a property or an indexer.
<code>nint</code>	Defines a native-sized integer data type.
<code>not</code>	Creates a pattern that matches when the negated pattern doesn't match.
<code>nuint</code>	Defines a native-sized unsigned integer data type.
<code>or</code>	Creates a pattern that matches when either of nested patterns matches.
<code>partial</code>	Defines partial classes, structs, and interfaces throughout the same compilation unit.
<code>record</code>	Used to define a record type.
<code>remove</code>	Defines a custom event accessor that is invoked when client code unsubscribes from the event.
<code>set</code>	Defines an accessor method for a property or an indexer.
<code>value</code>	Used to set accessors and to add or remove event handlers.

KEYWORD	DESCRIPTION
var	Enables the type of a variable declared at method scope to be determined by the compiler.
when	Specifies a filter condition for a <code>catch</code> block or the <code>case</code> label of a <code>switch</code> statement.
where	Adds constraints to a generic declaration. (See also where).
yield	Used in an iterator block to return a value to the enumerator object or to signal the end of iteration.

All query keywords introduced in C# 3.0 are also contextual. For more information, see [Query Keywords \(LINQ\)](#).

See also

- [C# reference](#)
- [C# keywords](#)
- [C# operators and expressions](#)

add (C# Reference)

12/28/2021 • 2 minutes to read • [Edit Online](#)

The `add` contextual keyword is used to define a custom event accessor that is invoked when client code subscribes to your [event](#). If you supply a custom `add` accessor, you must also supply a [remove](#) accessor.

Example

The following example shows an event that has custom `add` and [remove](#) accessors. For the full example, see [How to implement interface events](#).

```
class Events : IDrawingObject
{
    event EventHandler PreDrawEvent;

    event EventHandler IDrawingObject.OnDraw
    {
        add => PreDrawEvent += value;
        remove => PreDrawEvent -= value;
    }
}
```

You do not typically need to provide your own custom event accessors. The accessors that are automatically generated by the compiler when you declare an event are sufficient for most scenarios.

See also

- [Events](#)

get (C# Reference)

12/28/2021 • 2 minutes to read • [Edit Online](#)

The `get` keyword defines an *accessor* method in a property or indexer that returns the property value or the indexer element. For more information, see [Properties](#), [Auto-Implemented Properties](#) and [Indexers](#).

The following example defines both a `get` and a `set` accessor for a property named `Seconds`. It uses a private field named `_seconds` to back the property value.

```
class TimePeriod
{
    private double _seconds;

    public double Seconds
    {
        get { return _seconds; }
        set { _seconds = value; }
    }
}
```

Often, the `get` accessor consists of a single statement that returns a value, as it did in the previous example. Starting with C# 7.0, you can implement the `get` accessor as an expression-bodied member. The following example implements both the `get` and the `set` accessor as expression-bodied members.

```
class TimePeriod
{
    private double _seconds;

    public double Seconds
    {
        get => _seconds;
        set => _seconds = value;
    }
}
```

For simple cases in which a property's `get` and `set` accessors perform no other operation than setting or retrieving a value in a private backing field, you can take advantage of the C# compiler's support for auto-implemented properties. The following example implements `Hours` as an auto-implemented property.

```
class TimePeriod2
{
    public double Hours { get; set; }
}
```

C# Language Specification

For more information, see the [C# Language Specification](#). The language specification is the definitive source for C# syntax and usage.

See also

- [C# Reference](#)
- [C# Programming Guide](#)
- [C# Keywords](#)
- [Properties](#)

init (C# Reference)

12/28/2021 • 2 minutes to read • [Edit Online](#)

In C# 9 and later, the `init` keyword defines an *accessor* method in a property or indexer. An init-only setter assigns a value to the property or the indexer element only during object construction. For more information and examples, see [Properties](#), [Auto-Implemented Properties](#), and [Indexers](#).

The following example defines both a `get` and an `init` accessor for a property named `Seconds`. It uses a private field named `_seconds` to back the property value.

```
class InitExample
{
    private double _seconds;

    public double Seconds
    {
        get { return _seconds; }
        init { _seconds = value; }
    }
}
```

Often, the `init` accessor consists of a single statement that assigns a value, as it did in the previous example. You can implement the `init` accessor as an expression-bodied member. The following example implements both the `get` and the `init` accessors as expression-bodied members.

```
class InitExampleExpressionBodied
{
    private double _seconds;

    public double Seconds
    {
        get => _seconds;
        init => _seconds = value;
    }
}
```

For simple cases in which a property's `get` and `init` accessors perform no other operation than setting or retrieving a value in a private backing field, you can take advantage of the C# compiler's support for auto-implemented properties. The following example implements `Hours` as an auto-implemented property.

```
class InitExampleAutoProperty
{
    public double Hours { get; init; }
}
```

C# language specification

For more information, see the [C# Language Specification](#). The language specification is the definitive source for C# syntax and usage.

See also

- [C# Reference](#)
- [C# Programming Guide](#)
- [C# Keywords](#)
- [Properties](#)

partial type (C# Reference)

12/28/2021 • 2 minutes to read • [Edit Online](#)

Partial type definitions allow for the definition of a class, struct, interface, or record to be split into multiple files.

In *File1.cs*:

```
namespace PC
{
    partial class A
    {
        int num = 0;
        void MethodA() { }
        partial void MethodC();
    }
}
```

In *File2.cs* the declaration:

```
namespace PC
{
    partial class A
    {
        void MethodB() { }
        partial void MethodC() { }
    }
}
```

Remarks

Splitting a class, struct or interface type over several files can be useful when you are working with large projects, or with automatically generated code such as that provided by the [Windows Forms Designer](#). A partial type may contain a [partial method](#). For more information, see [Partial Classes and Methods](#).

C# language specification

For more information, see the [C# Language Specification](#). The language specification is the definitive source for C# syntax and usage.

See also

- [C# Reference](#)
- [C# Programming Guide](#)
- [Modifiers](#)
- [Introduction to Generics](#)

partial method (C# Reference)

12/28/2021 • 2 minutes to read • [Edit Online](#)

A partial method has its signature defined in one part of a partial type, and its implementation defined in another part of the type. Partial methods enable class designers to provide method hooks, similar to event handlers, that developers may decide to implement or not. If the developer does not supply an implementation, the compiler removes the signature at compile time. The following conditions apply to partial methods:

- Declarations must begin with the contextual keyword [partial](#).
- Signatures in both parts of the partial type must match.

A partial method isn't required to have an implementation in the following cases:

- It doesn't have any accessibility modifiers (including the default [private](#)).
- It returns [void](#).
- It doesn't have any [out](#) parameters.
- It doesn't have any of the following modifiers [virtual](#), [override](#), [sealed](#), [new](#), or [extern](#).

Any method that doesn't conform to all those restrictions (for example, `public virtual partial void method()`), must provide an implementation.

The following example shows a partial method defined in two parts of a partial class:

```
namespace PM
{
    partial class A
    {
        partial void OnSomethingHappened(string s);
    }

    // This part can be in a separate file.
    partial class A
    {
        // Comment out this method and the program
        // will still compile.
        partial void OnSomethingHappened(String s)
        {
            Console.WriteLine("Something happened: {0}", s);
        }
    }
}
```

Partial methods can also be useful in combination with source generators. For example a regex could be defined using the following pattern:

```
[RegexGenerated("(dog|cat|fish)")]
partial bool IsPetMatch(string input);
```

For more information, see [Partial Classes and Methods](#).

See also

- [C# Reference](#)
- [partial type](#)

remove (C# Reference)

12/28/2021 • 2 minutes to read • [Edit Online](#)

The `remove` contextual keyword is used to define a custom event accessor that is invoked when client code unsubscribes from your [event](#). If you supply a custom `remove` accessor, you must also supply an `add` accessor.

Example

The following example shows an event with custom `add` and `remove` accessors. For the full example, see [How to implement interface events](#).

```
class Events : IDrawingObject
{
    event EventHandler PreDrawEvent;

    event EventHandler IDrawingObject.OnDraw
    {
        add => PreDrawEvent += value;
        remove => PreDrawEvent -= value;
    }
}
```

You do not typically need to provide your own custom event accessors. The accessors that are automatically generated by the compiler when you declare an event are sufficient for most scenarios.

See also

- [Events](#)

set (C# Reference)

12/28/2021 • 2 minutes to read • [Edit Online](#)

The `set` keyword defines an *accessor* method in a property or indexer that assigns a value to the property or the indexer element. For more information and examples, see [Properties](#), [Auto-Implemented Properties](#), and [Indexers](#).

The following example defines both a `get` and a `set` accessor for a property named `Seconds`. It uses a private field named `_seconds` to back the property value.

```
class TimePeriod
{
    private double _seconds;

    public double Seconds
    {
        get { return _seconds; }
        set { _seconds = value; }
    }
}
```

Often, the `set` accessor consists of a single statement that assigns a value, as it did in the previous example. Starting with C# 7.0, you can implement the `set` accessor as an expression-bodied member. The following example implements both the `get` and the `set` accessors as expression-bodied members.

```
class TimePeriod
{
    private double _seconds;

    public double Seconds
    {
        get => _seconds;
        set => _seconds = value;
    }
}
```

For simple cases in which a property's `get` and `set` accessors perform no other operation than setting or retrieving a value in a private backing field, you can take advantage of the C# compiler's support for auto-implemented properties. The following example implements `Hours` as an auto-implemented property.

```
class TimePeriod2
{
    public double Hours { get; set; }
}
```

C# language specification

For more information, see the [C# Language Specification](#). The language specification is the definitive source for C# syntax and usage.

See also

- [C# Reference](#)
- [C# Programming Guide](#)
- [C# Keywords](#)
- [Properties](#)

when (C# reference)

12/28/2021 • 2 minutes to read • [Edit Online](#)

You use the `when` contextual keyword to specify a filter condition in the following contexts:

- In the `catch` statement of a `try/catch` or `try/catch/finally` block.
- As a `case guard` in the `switch` statement.
- As a `case guard` in the `switch` expression.

`when` in a `catch` statement

Starting with C# 6, `when` can be used in a `catch` statement to specify a condition that must be true for the handler for a specific exception to execute. Its syntax is:

```
catch (ExceptionType [e]) when (expr)
```

where *expr* is an expression that evaluates to a Boolean value. If it returns `true`, the exception handler executes; if `false`, it does not.

The following example uses the `when` keyword to conditionally execute handlers for an `HttpRequestException` depending on the text of the exception message.

```
using System;
using System.Net.Http;
using System.Threading.Tasks;

class Program
{
    static void Main()
    {
        Console.WriteLine(MakeRequest().Result);
    }

    public static async Task<string> MakeRequest()
    {
        var client = new HttpClient();
        var streamTask = client.GetStringAsync("https://localhost:10000");
        try
        {
            var responseText = await streamTask;
            return responseText;
        }
        catch (HttpRequestException e) when (e.Message.Contains("301"))
        {
            return "Site Moved";
        }
        catch (HttpRequestException e) when (e.Message.Contains("404"))
        {
            return "Page Not Found";
        }
        catch (HttpRequestException e)
        {
            return e.Message;
        }
    }
}
```

See also

- [try/catch statement](#)
- [try/catch/finally statement](#)

value (C# Reference)

12/28/2021 • 2 minutes to read • [Edit Online](#)

The contextual keyword `value` is used in the `set` accessor in [property](#) and [indexer](#) declarations. It is similar to an input parameter of a method. The word `value` references the value that client code is attempting to assign to the property or indexer. In the following example, `MyDerivedClass` has a property called `Name` that uses the `value` parameter to assign a new string to the backing field `_name`. From the point of view of client code, the operation is written as a simple assignment.

```
class MyBaseClass
{
    // virtual auto-implemented property. Overrides can only
    // provide specialized behavior if they implement get and set accessors.
    public virtual string Name { get; set; }

    // ordinary virtual property with backing field
    private int _num;
    public virtual int Number
    {
        get { return _num; }
        set { _num = value; }
    }
}

class MyDerivedClass : MyBaseClass
{
    private string _name;

    // Override auto-implemented property with ordinary property
    // to provide specialized accessor behavior.
    public override string Name
    {
        get
        {
            return _name;
        }
        set
        {
            if (!string.IsNullOrEmpty(value))
            {
                _name = value;
            }
            else
            {
                _name = "Unknown";
            }
        }
    }
}
```

For more information, see the [Properties](#) and [Indexers](#) articles.

C# language specification

For more information, see the [C# Language Specification](#). The language specification is the definitive source for C# syntax and usage.

See also

- [C# Reference](#)
- [C# Programming Guide](#)
- [C# Keywords](#)

yield (C# Reference)

12/28/2021 • 5 minutes to read • [Edit Online](#)

When you use the `yield` [contextual keyword](#) in a statement, you indicate that the method, operator, or `get` accessor in which it appears is an iterator. Using `yield` to define an iterator removes the need for an explicit extra class (the class that holds the state for an enumeration, see [IEnumerator<T>](#) for an example) when you implement the [IEnumerable](#) and [IEnumerator](#) pattern for a custom collection type.

The following example shows the two forms of the `yield` statement.

```
yield return <expression>;  
yield break;
```

Remarks

You use a `yield return` statement to return each element one at a time.

The sequence returned from an iterator method can be consumed by using a [foreach](#) statement or LINQ query. Each iteration of the `foreach` loop calls the iterator method. When a `yield return` statement is reached in the iterator method, `expression` is returned, and the current location in code is retained. Execution is restarted from that location the next time that the iterator function is called.

When the iterator returns an [System.Collections.Generic.IAsyncEnumerable<T>](#), that sequence can be consumed asynchronously using an [await foreach](#) statement. The iteration of the loop is analogous to the `foreach` statement. The difference is that each iteration may be suspended for an asynchronous operation before returning the expression for the next element.

You can use a `yield break` statement to end the iteration.

For more information about iterators, see [Iterators](#).

Iterator methods and get accessors

The declaration of an iterator must meet the following requirements:

- The return type must be one of the following types:
 - [IAsyncEnumerable<T>](#)
 - [IEnumerable<T>](#)
 - [IEnumerable](#)
 - [IEnumerator<T>](#)
 - [IEnumerator](#)
- The declaration can't have any [in](#), [ref](#), or [out](#) parameters.

The `yield` type of an iterator that returns [IEnumerable](#) or [IEnumerator](#) is `object`. If the iterator returns [IEnumerable<T>](#) or [IEnumerator<T>](#), there must be an implicit conversion from the type of the expression in the `yield return` statement to the generic type parameter.

You can't include a `yield return` or `yield break` statement in:

- [Lambda expressions](#) and [anonymous methods](#).
- Methods that contain unsafe blocks. For more information, see [unsafe](#).

Exception handling

A `yield return` statement can't be located in a try-catch block. A `yield return` statement can be located in the try block of a try-finally statement.

A `yield break` statement can be located in a try block or a catch block but not a finally block.

If the `foreach` or `await foreach` body (outside of the iterator method) throws an exception, a `finally` block in the iterator method is executed.

Technical implementation

The following code returns an `IEnumerable<string>` from an iterator method and then iterates through its elements.

```
IEnumerable<string> elements = MyIteratorMethod();
foreach (string element in elements)
{
    ...
}
```

The call to `MyIteratorMethod` doesn't execute the body of the method. Instead the call returns an `IEnumerable<string>` into the `elements` variable.

On an iteration of the `foreach` loop, the `MoveNext` method is called for `elements`. This call executes the body of `MyIteratorMethod` until the next `yield return` statement is reached. The expression returned by the `yield return` statement determines not only the value of the `element` variable for consumption by the loop body but also the `Current` property of `elements`, which is an `IEnumerator<string>`.

On each subsequent iteration of the `foreach` loop, the execution of the iterator body continues from where it left off, again stopping when it reaches a `yield return` statement. The `foreach` loop completes when the end of the iterator method or a `yield break` statement is reached.

The following code returns an `IAsyncEnumerable<string>` from an iterator method and then iterates through its elements.

```
IAsyncEnumerable<string> elements = MyAsyncIteratorMethod();
await foreach (string element in elements)
{
    // ...
}
```

On an iteration of the `await foreach` loop, the `IAsyncEnumerator<T>.MoveNextAsync` method is called for `elements`. The `System.Threading.Tasks.ValueTask<TResult>` return by `MoveNext` completes when the next `yield return` is reached.

On each subsequent iteration of the `await foreach` loop, the execution of the iterator body continues from where it left off, again stopping when it reaches a `yield return` statement. The `await foreach` loop completes when the end of the iterator method or a `yield break` statement is reached.

Examples

The following example has a `yield return` statement that's inside a `for` loop. Each iteration of the `foreach` statement body in the `Main` method creates a call to the `Power` iterator function. Each call to the iterator function proceeds to the next execution of the `yield return` statement, which occurs during the next iteration of

the `for` loop.

The return type of the iterator method is `IEnumerable`, which is an iterator interface type. When the iterator method is called, it returns an enumerable object that contains the powers of a number.

```
public class PowersOf2
{
    static void Main()
    {
        // Display powers of 2 up to the exponent of 8:
        foreach (int i in Power(2, 8))
        {
            Console.Write("{0} ", i);
        }
    }

    public static System.Collections.Generic.IEnumerable<int> Power(int number, int exponent)
    {
        int result = 1;

        for (int i = 0; i < exponent; i++)
        {
            result = result * number;
            yield return result;
        }
    }

    // Output: 2 4 8 16 32 64 128 256
}
```

The following example demonstrates a `get` accessor that is an iterator. In the example, each `yield return` statement returns an instance of a user-defined class.


```

public static class GalaxyClass
{
    public static void ShowGalaxies()
    {
        var theGalaxies = new Galaxies();
        foreach (Galaxy theGalaxy in theGalaxies.NextGalaxy)
        {
            Debug.WriteLine(theGalaxy.Name + " " + theGalaxy.MegaLightYears.ToString());
        }
    }

    public class Galaxies
    {
        public System.Collections.Generic.IEnumerable<Galaxy> NextGalaxy
        {
            get
            {
                yield return new Galaxy { Name = "Tadpole", MegaLightYears = 400 };
                yield return new Galaxy { Name = "Pinwheel", MegaLightYears = 25 };
                yield return new Galaxy { Name = "Milky Way", MegaLightYears = 0 };
                yield return new Galaxy { Name = "Andromeda", MegaLightYears = 3 };
            }
        }
    }

    public class Galaxy
    {
        public String Name { get; set; }
        public int MegaLightYears { get; set; }
    }
}

```

C# language specification

For more information, see the [C# Language Specification](#). The language specification is the definitive source for C# syntax and usage.

See also

- [C# Reference](#)
- [C# Programming Guide](#)
- [foreach, in](#)
- [Iterators](#)

Query keywords (C# Reference)

12/28/2021 • 2 minutes to read • [Edit Online](#)

This section contains the contextual keywords used in query expressions.

In this section

CLAUSE	DESCRIPTION
from	Specifies a data source and a range variable (similar to an iteration variable).
where	Filters source elements based on one or more Boolean expressions separated by logical AND and OR operators (<code>&&</code> or <code> </code>).
select	Specifies the type and shape that the elements in the returned sequence will have when the query is executed.
group	Groups query results according to a specified key value.
into	Provides an identifier that can serve as a reference to the results of a join, group or select clause.
orderby	Sorts query results in ascending or descending order based on the default comparer for the element type.
join	Joins two data sources based on an equality comparison between two specified matching criteria.
let	Introduces a range variable to store sub-expression results in a query expression.
in	Contextual keyword in a join clause.
on	Contextual keyword in a join clause.
equals	Contextual keyword in a join clause.
by	Contextual keyword in a group clause.
ascending	Contextual keyword in an orderby clause.
descending	Contextual keyword in an orderby clause.

See also

- [C# Keywords](#)
- [LINQ \(Language-Integrated Query\)](#)
- [LINQ in C#](#)

from clause (C# Reference)

12/28/2021 • 5 minutes to read • [Edit Online](#)

A query expression must begin with a `from` clause. Additionally, a query expression can contain sub-queries, which also begin with a `from` clause. The `from` clause specifies the following:

- The data source on which the query or sub-query will be run.
- A local *range variable* that represents each element in the source sequence.

Both the range variable and the data source are strongly typed. The data source referenced in the `from` clause must have a type of `IEnumerable`, `IEnumerable<T>`, or a derived type such as `IQueryable<T>`.

In the following example, `numbers` is the data source and `num` is the range variable. Note that both variables are strongly typed even though the `var` keyword is used.

```
class LowNums
{
    static void Main()
    {
        // A simple data source.
        int[] numbers = { 5, 4, 1, 3, 9, 8, 6, 7, 2, 0 };

        // Create the query.
        // lowNums is an IEnumerable<int>
        var lowNums = from num in numbers
                      where num < 5
                      select num;

        // Execute the query.
        foreach (int i in lowNums)
        {
            Console.Write(i + " ");
        }
    }
}
// Output: 4 1 3 2 0
```

The range variable

The compiler infers the type of the range variable when the data source implements `IEnumerable<T>`. For example, if the source has a type of `IEnumerable<Customer>`, then the range variable is inferred to be `Customer`. The only time that you must specify the type explicitly is when the source is a non-generic `IEnumerable` type such as `ArrayList`. For more information, see [How to query an ArrayList with LINQ](#).

In the previous example `num` is inferred to be of type `int`. Because the range variable is strongly typed, you can call methods on it or use it in other operations. For example, instead of writing `select num`, you could write `select num.ToString()` to cause the query expression to return a sequence of strings instead of integers. Or you could write `select num + 10` to cause the expression to return the sequence 14, 11, 13, 12, 10. For more information, see [select clause](#).

The range variable is like an iteration variable in a `foreach` statement except for one very important difference: a range variable never actually stores data from the source. It's just a syntactic convenience that enables the query to describe what will occur when the query is executed. For more information, see [Introduction to LINQ Queries \(C#\)](#).

Compound from clauses

In some cases, each element in the source sequence may itself be either a sequence or contain a sequence. For example, your data source may be an `IEnumerable<Student>` where each student object in the sequence contains a list of test scores. To access the inner list within each `Student` element, you can use compound `from` clauses. The technique is like using nested `foreach` statements. You can add `where` or `orderby` clauses to either `from` clause to filter the results. The following example shows a sequence of `Student` objects, each of which contains an inner `List` of integers representing test scores. To access the inner list, use a compound `from` clause. You can insert clauses between the two `from` clauses if necessary.

```

class CompoundFrom
{
    // The element type of the data source.
    public class Student
    {
        public string LastName { get; set; }
        public List<int> Scores {get; set;}
    }

    static void Main()
    {
        // Use a collection initializer to create the data source. Note that
        // each element in the list contains an inner sequence of scores.
        List<Student> students = new List<Student>
        {
            new Student {LastName="Omelchenko", Scores= new List<int> {97, 72, 81, 60}},
            new Student {LastName="O'Donnell", Scores= new List<int> {75, 84, 91, 39}},
            new Student {LastName="Mortensen", Scores= new List<int> {88, 94, 65, 85}},
            new Student {LastName="Garcia", Scores= new List<int> {97, 89, 85, 82}},
            new Student {LastName="Beebe", Scores= new List<int> {35, 72, 91, 70}}
        };

        // Use a compound from to access the inner sequence within each element.
        // Note the similarity to a nested foreach statement.
        var scoreQuery = from student in students
                        from score in student.Scores
                        where score > 90
                        select new { Last = student.LastName, score };

        // Execute the queries.
        Console.WriteLine("scoreQuery:");
        // Rest the mouse pointer on scoreQuery in the following line to
        // see its type. The type is IEnumerable<'a>, where 'a is an
        // anonymous type defined as new {string Last, int score}. That is,
        // each instance of this anonymous type has two members, a string
        // (Last) and an int (score).
        foreach (var student in scoreQuery)
        {
            Console.WriteLine("{0} Score: {1}", student.Last, student.score);
        }

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
/*
scoreQuery:
Omelchenko Score: 97
O'Donnell Score: 91
Mortensen Score: 94
Garcia Score: 97
Beebe Score: 91
*/

```

Using Multiple from Clauses to Perform Joins

A compound `from` clause is used to access inner collections in a single data source. However, a query can also contain multiple `from` clauses that generate supplemental queries from independent data sources. This technique enables you to perform certain types of join operations that are not possible by using the [join clause](#).

The following example shows how two `from` clauses can be used to form a complete cross join of two data sources.

```

class CompoundFrom2
{
    static void Main()
    {
        char[] upperCase = { 'A', 'B', 'C' };
        char[] lowerCase = { 'x', 'y', 'z' };

        // The type of joinQuery1 is IEnumerable<'a>, where 'a
        // indicates an anonymous type. This anonymous type has two
        // members, upper and lower, both of type char.
        var joinQuery1 =
            from upper in upperCase
            from lower in lowerCase
            select new { upper, lower };

        // The type of joinQuery2 is IEnumerable<'a>, where 'a
        // indicates an anonymous type. This anonymous type has two
        // members, upper and lower, both of type char.
        var joinQuery2 =
            from lower in lowerCase
            where lower != 'x'
            from upper in upperCase
            select new { lower, upper };

        // Execute the queries.
        Console.WriteLine("Cross join:");
        // Rest the mouse pointer on joinQuery1 to verify its type.
        foreach (var pair in joinQuery1)
        {
            Console.WriteLine("{0} is matched to {1}", pair.upper, pair.lower);
        }

        Console.WriteLine("Filtered non-equijoin:");
        // Rest the mouse pointer over joinQuery2 to verify its type.
        foreach (var pair in joinQuery2)
        {
            Console.WriteLine("{0} is matched to {1}", pair.lower, pair.upper);
        }

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
/* Output:
Cross join:
A is matched to x
A is matched to y
A is matched to z
B is matched to x
B is matched to y
B is matched to z
C is matched to x
C is matched to y
C is matched to z
Filtered non-equijoin:
y is matched to A
y is matched to B
y is matched to C
z is matched to A
z is matched to B
z is matched to C
*/

```

See also

- [Query Keywords \(LINQ\)](#)
- [Language Integrated Query \(LINQ\)](#)

where clause (C# Reference)

12/28/2021 • 3 minutes to read • [Edit Online](#)

The `where` clause is used in a query expression to specify which elements from the data source will be returned in the query expression. It applies a Boolean condition (*predicate*) to each source element (referenced by the range variable) and returns those for which the specified condition is true. A single query expression may contain multiple `where` clauses and a single clause may contain multiple predicate subexpressions.

Example 1

In the following example, the `where` clause filters out all numbers except those that are less than five. If you remove the `where` clause, all numbers from the data source would be returned. The expression `num < 5` is the predicate that is applied to each element.

```
class WhereSample
{
    static void Main()
    {
        // Simple data source. Arrays support IEnumerable<T>.
        int[] numbers = { 5, 4, 1, 3, 9, 8, 6, 7, 2, 0 };

        // Simple query with one predicate in where clause.
        var queryLowNums =
            from num in numbers
            where num < 5
            select num;

        // Execute the query.
        foreach (var s in queryLowNums)
        {
            Console.Write(s.ToString() + " ");
        }
    }
}
//Output: 4 1 3 2 0
```

Example 2

Within a single `where` clause, you can specify as many predicates as necessary by using the `&&` and `||` operators. In the following example, the query specifies two predicates in order to select only the even numbers that are less than five.


```

class WhereSample2
{
    static void Main()
    {
        // Data source.
        int[] numbers = { 5, 4, 1, 3, 9, 8, 6, 7, 2, 0 };

        // Create the query with two predicates in where clause.
        var queryLowNums2 =
            from num in numbers
            where num < 5 && num % 2 == 0
            select num;

        // Execute the query
        foreach (var s in queryLowNums2)
        {
            Console.Write(s.ToString() + " ");
        }
        Console.WriteLine();

        // Create the query with two where clause.
        var queryLowNums3 =
            from num in numbers
            where num < 5
            where num % 2 == 0
            select num;

        // Execute the query
        foreach (var s in queryLowNums3)
        {
            Console.Write(s.ToString() + " ");
        }
    }
}
// Output:
// 4 2 0
// 4 2 0

```

Example 3

A `where` clause may contain one or more methods that return Boolean values. In the following example, the `where` clause uses a method to determine whether the current value of the range variable is even or odd.

```

class WhereSample3
{
    static void Main()
    {
        // Data source
        int[] numbers = { 5, 4, 1, 3, 9, 8, 6, 7, 2, 0 };

        // Create the query with a method call in the where clause.
        // Note: This won't work in LINQ to SQL unless you have a
        // stored procedure that is mapped to a method by this name.
        var queryEvenNums =
            from num in numbers
            where IsEven(num)
            select num;

        // Execute the query.
        foreach (var s in queryEvenNums)
        {
            Console.Write(s.ToString() + " ");
        }

        // Method may be instance method or static method.
        static bool IsEven(int i)
        {
            return i % 2 == 0;
        }
    }
}
//Output: 4 8 6 2 0

```

Remarks

The `where` clause is a filtering mechanism. It can be positioned almost anywhere in a query expression, except it cannot be the first or last clause. A `where` clause may appear either before or after a `group` clause depending on whether you have to filter the source elements before or after they are grouped.

If a specified predicate is not valid for the elements in the data source, a compile-time error will result. This is one benefit of the strong type-checking provided by LINQ.

At compile time the `where` keyword is converted into a call to the [Where](#) Standard Query Operator method.

See also

- [Query Keywords \(LINQ\)](#)
- [from clause](#)
- [select clause](#)
- [Filtering Data](#)
- [LINQ in C#](#)
- [Language Integrated Query \(LINQ\)](#)

select clause (C# Reference)

12/28/2021 • 6 minutes to read • [Edit Online](#)

In a query expression, the `select` clause specifies the type of values that will be produced when the query is executed. The result is based on the evaluation of all the previous clauses and on any expressions in the `select` clause itself. A query expression must terminate with either a `select` clause or a `group` clause.

The following example shows a simple `select` clause in a query expression.

```
class SelectSample1
{
    static void Main()
    {
        //Create the data source
        List<int> Scores = new List<int>() { 97, 92, 81, 60 };

        // Create the query.
        IEnumerable<int> queryHighScores =
            from score in Scores
            where score > 80
            select score;

        // Execute the query.
        foreach (int i in queryHighScores)
        {
            Console.Write(i + " ");
        }
    }
}
//Output: 97 92 81
```

The type of the sequence produced by the `select` clause determines the type of the query variable `queryHighScores`. In the simplest case, the `select` clause just specifies the range variable. This causes the returned sequence to contain elements of the same type as the data source. For more information, see [Type Relationships in LINQ Query Operations](#). However, the `select` clause also provides a powerful mechanism for transforming (or *projecting*) source data into new types. For more information, see [Data Transformations with LINQ \(C#\)](#).

Example

The following example shows all the different forms that a `select` clause may take. In each query, note the relationship between the `select` clause and the type of the *query variable* (`studentQuery1`, `studentQuery2`, and so on).

```
class SelectSample2
{
    // Define some classes
    public class Student
    {
        public string First { get; set; }
        public string Last { get; set; }
        public int ID { get; set; }
        public List<int> Scores;
        public ContactInfo GetContactInfo(SelectSample2 app, int id)
        {
            ContactInfo cInfo =
```

```

        (from ci in app.contactList
         where ci.ID == id
         select ci)
        .FirstOrDefault();

        return cInfo;
    }

    public override string ToString()
    {
        return First + " " + Last + ":" + ID;
    }
}

public class ContactInfo
{
    public int ID { get; set; }
    public string Email { get; set; }
    public string Phone { get; set; }
    public override string ToString() { return Email + "," + Phone; }
}

public class ScoreInfo
{
    public double Average { get; set; }
    public int ID { get; set; }
}

// The primary data source
List<Student> students = new List<Student>()
{
    new Student {First="Svetlana", Last="Omelchenko", ID=111, Scores= new List<int>() {97, 92, 81,
60}},
    new Student {First="Claire", Last="O'Donnell", ID=112, Scores= new List<int>() {75, 84, 91,
39}},
    new Student {First="Sven", Last="Mortensen", ID=113, Scores= new List<int>() {88, 94, 65, 91}},
    new Student {First="Cesar", Last="Garcia", ID=114, Scores= new List<int>() {97, 89, 85, 82}},
};

// Separate data source for contact info.
List<ContactInfo> contactList = new List<ContactInfo>()
{
    new ContactInfo {ID=111, Email="SvetlanO@Contoso.com", Phone="206-555-0108"},
    new ContactInfo {ID=112, Email="ClaireO@Contoso.com", Phone="206-555-0298"},
    new ContactInfo {ID=113, Email="SvenMort@Contoso.com", Phone="206-555-1130"},
    new ContactInfo {ID=114, Email="CesarGar@Contoso.com", Phone="206-555-0521"}
};

static void Main(string[] args)
{
    SelectSample2 app = new SelectSample2();

    // Produce a filtered sequence of unmodified Students.
    IEnumerable<Student> studentQuery1 =
        from student in app.students
        where student.ID > 111
        select student;

    Console.WriteLine("Query1: select range_variable");
    foreach (Student s in studentQuery1)
    {
        Console.WriteLine(s.ToString());
    }

    // Produce a filtered sequence of elements that contain
    // only one property of each Student.
    IEnumerable<String> studentQuery2 =
        from student in app.students
        where student.ID > 111

```

```

        select student.Last;

Console.WriteLine("\r\n studentQuery2: select range_variable.Property");
foreach (string s in studentQuery2)
{
    Console.WriteLine(s);
}

// Produce a filtered sequence of objects created by
// a method call on each Student.
IEnumerable<ContactInfo> studentQuery3 =
    from student in app.students
    where student.ID > 111
    select student.GetContactInfo(app, student.ID);

Console.WriteLine("\r\n studentQuery3: select range_variable.Method");
foreach (ContactInfo ci in studentQuery3)
{
    Console.WriteLine(ci.ToString());
}

// Produce a filtered sequence of ints from
// the internal array inside each Student.
IEnumerable<int> studentQuery4 =
    from student in app.students
    where student.ID > 111
    select student.Scores[0];

Console.WriteLine("\r\n studentQuery4: select range_variable[index]");
foreach (int i in studentQuery4)
{
    Console.WriteLine("First score = {0}", i);
}

// Produce a filtered sequence of doubles
// that are the result of an expression.
IEnumerable<double> studentQuery5 =
    from student in app.students
    where student.ID > 111
    select student.Scores[0] * 1.1;

Console.WriteLine("\r\n studentQuery5: select expression");
foreach (double d in studentQuery5)
{
    Console.WriteLine("Adjusted first score = {0}", d);
}

// Produce a filtered sequence of doubles that are
// the result of a method call.
IEnumerable<double> studentQuery6 =
    from student in app.students
    where student.ID > 111
    select student.Scores.Average();

Console.WriteLine("\r\n studentQuery6: select expression2");
foreach (double d in studentQuery6)
{
    Console.WriteLine("Average = {0}", d);
}

// Produce a filtered sequence of anonymous types
// that contain only two properties from each Student.
var studentQuery7 =
    from student in app.students
    where student.ID > 111
    select new { student.First, student.Last };

Console.WriteLine("\r\n studentQuery7: select new anonymous type");
foreach (var item in studentQuery7)

```

```

    {
        Console.WriteLine("{0}, {1}", item.Last, item.First);
    }

    // Produce a filtered sequence of named objects that contain
    // a method return value and a property from each Student.
    // Use named types if you need to pass the query variable
    // across a method boundary.
    IEnumerable<ScoreInfo> studentQuery8 =
        from student in app.students
        where student.ID > 111
        select new ScoreInfo
        {
            Average = student.Scores.Average(),
            ID = student.ID
        };

    Console.WriteLine("\r\n studentQuery8: select new named type");
    foreach (ScoreInfo si in studentQuery8)
    {
        Console.WriteLine("ID = {0}, Average = {1}", si.ID, si.Average);
    }

    // Produce a filtered sequence of students who appear on a contact list
    // and whose average is greater than 85.
    IEnumerable<ContactInfo> studentQuery9 =
        from student in app.students
        where student.Scores.Average() > 85
        join ci in app.contactList on student.ID equals ci.ID
        select ci;

    Console.WriteLine("\r\n studentQuery9: select result of join clause");
    foreach (ContactInfo ci in studentQuery9)
    {
        Console.WriteLine("ID = {0}, Email = {1}", ci.ID, ci.Email);
    }

    // Keep the console window open in debug mode
    Console.WriteLine("Press any key to exit.");
    Console.ReadKey();
}
}

/* Output
Query1: select range_variable
Claire O'Donnell:112
Sven Mortensen:113
Cesar Garcia:114

studentQuery2: select range_variable.Property
O'Donnell
Mortensen
Garcia

studentQuery3: select range_variable.Method
ClaireO@Contoso.com,206-555-0298
SvenMort@Contoso.com,206-555-1130
CesarGar@Contoso.com,206-555-0521

studentQuery4: select range_variable[index]
First score = 75
First score = 88
First score = 97

studentQuery5: select expression
Adjusted first score = 82.5
Adjusted first score = 96.8
Adjusted first score = 106.7

studentQuery6: select expression?

```

```

studentQuery6: select expression2
Average = 72.25
Average = 84.5
Average = 88.25

studentQuery7: select new anonymous type
O'Donnell, Claire
Mortensen, Sven
Garcia, Cesar

studentQuery8: select new named type
ID = 112, Average = 72.25
ID = 113, Average = 84.5
ID = 114, Average = 88.25

studentQuery9: select result of join clause
ID = 114, Email = CesarGar@Contoso.com
*/

```

As shown in `studentQuery8` in the previous example, sometimes you might want the elements of the returned sequence to contain only a subset of the properties of the source elements. By keeping the returned sequence as small as possible you can reduce the memory requirements and increase the speed of the execution of the query. You can accomplish this by creating an anonymous type in the `select` clause and using an object initializer to initialize it with the appropriate properties from the source element. For an example of how to do this, see [Object and Collection Initializers](#).

Remarks

At compile time, the `select` clause is translated to a method call to the [Select](#) standard query operator.

See also

- [C# Reference](#)
- [Query Keywords \(LINQ\)](#)
- [from clause](#)
- [partial \(Method\) \(C# Reference\)](#)
- [Anonymous Types](#)
- [LINQ in C#](#)
- [Language Integrated Query \(LINQ\)](#)

group clause (C# Reference)

12/28/2021 • 8 minutes to read • [Edit Online](#)

The `group` clause returns a sequence of `IGrouping<TKey,TElement>` objects that contain zero or more items that match the key value for the group. For example, you can group a sequence of strings according to the first letter in each string. In this case, the first letter is the key and has a type `char`, and is stored in the `Key` property of each `IGrouping<TKey,TElement>` object. The compiler infers the type of the key.

You can end a query expression with a `group` clause, as shown in the following example:

```
// Query variable is an IEnumerable<IGrouping<char, Student>>
var studentQuery1 =
    from student in students
    group student by student.Last[0];
```

If you want to perform additional query operations on each group, you can specify a temporary identifier by using the `into` contextual keyword. When you use `into`, you must continue with the query, and eventually end it with either a `select` statement or another `group` clause, as shown in the following excerpt:

```
// Group students by the first letter of their last name
// Query variable is an IEnumerable<IGrouping<char, Student>>
var studentQuery2 =
    from student in students
    group student by student.Last[0] into g
    orderby g.Key
    select g;
```

More complete examples of the use of `group` with and without `into` are provided in the Example section of this article.

Enumerating the results of a group query

Because the `IGrouping<TKey,TElement>` objects produced by a `group` query are essentially a list of lists, you must use a nested `foreach` loop to access the items in each group. The outer loop iterates over the group keys, and the inner loop iterates over each item in the group itself. A group may have a key but no elements. The following is the `foreach` loop that executes the query in the previous code examples:

```
// Iterate group items with a nested foreach. This IGrouping encapsulates
// a sequence of Student objects, and a Key of type char.
// For convenience, var can also be used in the foreach statement.
foreach (IGrouping<char, Student> studentGroup in studentQuery2)
{
    Console.WriteLine(studentGroup.Key);
    // Explicit type for student could also be used here.
    foreach (var student in studentGroup)
    {
        Console.WriteLine("  {0}, {1}", student.Last, student.First);
    }
}
```

Key types

Group keys can be any type, such as a string, a built-in numeric type, or a user-defined named type or anonymous type.

Grouping by string

The previous code examples used a `char`. A string key could easily have been specified instead, for example the complete last name:

```
// Same as previous example except we use the entire last name as a key.  
// Query variable is an IEnumerable<IGrouping<string, Student>>  
var studentQuery3 =  
    from student in students  
    group student by student.Last;
```

Grouping by bool

The following example shows the use of a bool value for a key to divide the results into two groups. Note that the value is produced by a sub-expression in the `group` clause.

```

class GroupSample1
{
    // The element type of the data source.
    public class Student
    {
        public string First { get; set; }
        public string Last { get; set; }
        public int ID { get; set; }
        public List<int> Scores;
    }

    public static List<Student> GetStudents()
    {
        // Use a collection initializer to create the data source. Note that each element
        // in the list contains an inner sequence of scores.
        List<Student> students = new List<Student>
        {
            new Student {First="Svetlana", Last="Omelchenko", ID=111, Scores= new List<int> {97, 72, 81,
60}},
            new Student {First="Claire", Last="O'Donnell", ID=112, Scores= new List<int> {75, 84, 91, 39}},
            new Student {First="Sven", Last="Mortensen", ID=113, Scores= new List<int> {99, 89, 91, 95}},
            new Student {First="Cesar", Last="Garcia", ID=114, Scores= new List<int> {72, 81, 65, 84}},
            new Student {First="Debra", Last="Garcia", ID=115, Scores= new List<int> {97, 89, 85, 82}}
        };

        return students;
    }

    static void Main()
    {
        // Obtain the data source.
        List<Student> students = GetStudents();

        // Group by true or false.
        // Query variable is an IEnumerable<IGrouping<bool, Student>>
        var booleanGroupQuery =
            from student in students
            group student by student.Scores.Average() >= 80; //pass or fail!

        // Execute the query and access items in each group
        foreach (var studentGroup in booleanGroupQuery)
        {
            Console.WriteLine(studentGroup.Key == true ? "High averages" : "Low averages");
            foreach (var student in studentGroup)
            {
                Console.WriteLine("    {0}, {1}:{2}", student.Last, student.First, student.Scores.Average());
            }
        }

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
/* Output:
Low averages
Omelchenko, Svetlana:77.5
O'Donnell, Claire:72.25
Garcia, Cesar:75.5
High averages
Mortensen, Sven:93.5
Garcia, Debra:88.25
*/

```

Grouping by numeric range

The next example uses an expression to create numeric group keys that represent a percentile range. Note the

use of [let](#) as a convenient location to store a method call result, so that you don't have to call the method two times in the `group` clause. For more information about how to safely use methods in query expressions, see [Handle exceptions in query expressions](#).

```
class GroupSample2
{
    // The element type of the data source.
    public class Student
    {
        public string First { get; set; }
        public string Last { get; set; }
        public int ID { get; set; }
        public List<int> Scores;
    }

    public static List<Student> GetStudents()
    {
        // Use a collection initializer to create the data source. Note that each element
        // in the list contains an inner sequence of scores.
        List<Student> students = new List<Student>
        {
            new Student {First="Svetlana", Last="Omelchenko", ID=111, Scores= new List<int> {97, 72, 81,
60}},
            new Student {First="Claire", Last="O'Donnell", ID=112, Scores= new List<int> {75, 84, 91, 39}},
            new Student {First="Sven", Last="Mortensen", ID=113, Scores= new List<int> {99, 89, 91, 95}},
            new Student {First="Cesar", Last="Garcia", ID=114, Scores= new List<int> {72, 81, 65, 84}},
            new Student {First="Debra", Last="Garcia", ID=115, Scores= new List<int> {97, 89, 85, 82}}
        };

        return students;
    }

    // This method groups students into percentile ranges based on their
    // grade average. The Average method returns a double, so to produce a whole
    // number it is necessary to cast to int before dividing by 10.
    static void Main()
    {
        // Obtain the data source.
        List<Student> students = GetStudents();

        // Write the query.
        var studentQuery =
            from student in students
            let avg = (int)student.Scores.Average()
            group student by (avg / 10) into g
            orderby g.Key
            select g;

        // Execute the query.
        foreach (var studentGroup in studentQuery)
        {
            int temp = studentGroup.Key * 10;
            Console.WriteLine("Students with an average between {0} and {1}", temp, temp + 10);
            foreach (var student in studentGroup)
            {
                Console.WriteLine("    {0}, {1}:{2}", student.Last, student.First, student.Scores.Average());
            }
        }

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}

/* Output:
    Students with an average between 70 and 80
    Omelchenko, Svetlana:77.5
```

```
O'Donnell, Claire:72.25
Garcia, Cesar:75.5
Students with an average between 80 and 90
Garcia, Debra:88.25
Students with an average between 90 and 100
Mortensen, Sven:93.5

*/
```

Grouping by composite keys

Use a composite key when you want to group elements according to more than one key. You create a composite key by using an anonymous type or a named type to hold the key element. In the following example, assume that a class `Person` has been declared with members named `surname` and `city`. The `group` clause causes a separate group to be created for each set of persons with the same last name and the same city.

```
group person by new {name = person.surname, city = person.city};
```

Use a named type if you must pass the query variable to another method. Create a special class using auto-implemented properties for the keys, and then override the [Equals](#) and [GetHashCode](#) methods. You can also use a struct, in which case you do not strictly have to override those methods. For more information see [How to implement a lightweight class with auto-implemented properties](#) and [How to query for duplicate files in a directory tree](#). The latter article has a code example that demonstrates how to use a composite key with a named type.

Example 1

The following example shows the standard pattern for ordering source data into groups when no additional query logic is applied to the groups. This is called a grouping without a continuation. The elements in an array of strings are grouped according to their first letter. The result of the query is an `IGrouping<TKey,TElement>` type that contains a public `Key` property of type `char` and an `IEnumerable<T>` collection that contains each item in the grouping.

The result of a `group` clause is a sequence of sequences. Therefore, to access the individual elements within each returned group, use a nested `foreach` loop inside the loop that iterates the group keys, as shown in the following example.

```

class GroupExample1
{
    static void Main()
    {
        // Create a data source.
        string[] words = { "blueberry", "chimpanzee", "abacus", "banana", "apple", "cheese" };

        // Create the query.
        var wordGroups =
            from w in words
            group w by w[0];

        // Execute the query.
        foreach (var wordGroup in wordGroups)
        {
            Console.WriteLine("Words that start with the letter '{0}':", wordGroup.Key);
            foreach (var word in wordGroup)
            {
                Console.WriteLine(word);
            }
        }

        // Keep the console window open in debug mode
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
/* Output:
Words that start with the letter 'b':
blueberry
banana
Words that start with the letter 'c':
chimpanzee
cheese
Words that start with the letter 'a':
abacus
apple
*/

```

Example 2

This example shows how to perform additional logic on the groups after you have created them, by using a *continuation* with `into`. For more information, see [into](#). The following example queries each group to select only those whose key value is a vowel.

```

class GroupClauseExample2
{
    static void Main()
    {
        // Create the data source.
        string[] words2 = { "blueberry", "chimpanzee", "abacus", "banana", "apple", "cheese", "elephant",
"umbrella", "anteater" };

        // Create the query.
        var wordGroups2 =
            from w in words2
            group w by w[0] into grps
            where (grps.Key == 'a' || grps.Key == 'e' || grps.Key == 'i'
                || grps.Key == 'o' || grps.Key == 'u')
            select grps;

        // Execute the query.
        foreach (var wordGroup in wordGroups2)
        {
            Console.WriteLine("Groups that start with a vowel: {0}", wordGroup.Key);
            foreach (var word in wordGroup)
            {
                Console.WriteLine("    {0}", word);
            }
        }

        // Keep the console window open in debug mode
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
/* Output:
    Groups that start with a vowel: a
        abacus
        apple
        anteater
    Groups that start with a vowel: e
        elephant
    Groups that start with a vowel: u
        umbrella
*/

```

Remarks

At compile time, `group` clauses are translated into calls to the [GroupBy](#) method.

See also

- [IGrouping<TKey,TElement>](#)
- [GroupBy](#)
- [ThenBy](#)
- [ThenByDescending](#)
- [Query Keywords](#)
- [Language Integrated Query \(LINQ\)](#)
- [Create a nested group](#)
- [Group query results](#)
- [Perform a subquery on a grouping operation](#)

into (C# Reference)

12/28/2021 • 2 minutes to read • [Edit Online](#)

The `into` contextual keyword can be used to create a temporary identifier to store the results of a [group](#), [join](#) or [select](#) clause into a new identifier. This identifier can itself be a generator for additional query commands. When used in a `group` or `select` clause, the use of the new identifier is sometimes referred to as a *continuation*.

Example

The following example shows the use of the `into` keyword to enable a temporary identifier `fruitGroup` which has an inferred type of `IGrouping`. By using the identifier, you can invoke the [Count](#) method on each group and select only those groups that contain two or more words.

```
class IntoSample1
{
    static void Main()
    {
        // Create a data source.
        string[] words = { "apples", "blueberries", "oranges", "bananas", "apricots"};

        // Create the query.
        var wordGroups1 =
            from w in words
            group w by w[0] into fruitGroup
            where fruitGroup.Count() >= 2
            select new { FirstLetter = fruitGroup.Key, Words = fruitGroup.Count() };

        // Execute the query. Note that we only iterate over the groups,
        // not the items in each group
        foreach (var item in wordGroups1)
        {
            Console.WriteLine(" {0} has {1} elements.", item.FirstLetter, item.Words);
        }

        // Keep the console window open in debug mode
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
/* Output:
a has 2 elements.
b has 2 elements.
*/
```

The use of `into` in a `group` clause is only necessary when you want to perform additional query operations on each group. For more information, see [group clause](#).

For an example of the use of `into` in a `join` clause, see [join clause](#).

See also

- [Query Keywords \(LINQ\)](#)
- [LINQ in C#](#)
- [group clause](#)

orderby clause (C# Reference)

12/28/2021 • 2 minutes to read • [Edit Online](#)

In a query expression, the `orderby` clause causes the returned sequence or subsequence (group) to be sorted in either ascending or descending order. Multiple keys can be specified in order to perform one or more secondary sort operations. The sorting is performed by the default comparer for the type of the element. The default sort order is ascending. You can also specify a custom comparer. However, it is only available by using method-based syntax. For more information, see [Sorting Data](#).

Example 1

In the following example, the first query sorts the words in alphabetical order starting from A, and second query sorts the same words in descending order. (The `ascending` keyword is the default sort value and can be omitted.)


```

class OrderbySample1
{
    static void Main()
    {
        // Create a delicious data source.
        string[] fruits = { "cherry", "apple", "blueberry" };

        // Query for ascending sort.
        IEnumerable<string> sortAscendingQuery =
            from fruit in fruits
            orderby fruit //"ascending" is default
            select fruit;

        // Query for descending sort.
        IEnumerable<string> sortDescendingQuery =
            from w in fruits
            orderby w descending
            select w;

        // Execute the query.
        Console.WriteLine("Ascending:");
        foreach (string s in sortAscendingQuery)
        {
            Console.WriteLine(s);
        }

        // Execute the query.
        Console.WriteLine(Environment.NewLine + "Descending:");
        foreach (string s in sortDescendingQuery)
        {
            Console.WriteLine(s);
        }

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
/* Output:
Ascending:
apple
blueberry
cherry

Descending:
cherry
blueberry
apple
*/

```

Example 2

The following example performs a primary sort on the students' last names, and then a secondary sort on their first names.

```

class OrderbySample2
{
    // The element type of the data source.
    public class Student
    {
        public string First { get; set; }
        public string Last { get; set; }
        public int ID { get; set; }
    }
}

```

```

public static List<Student> GetStudents()
{
    // Use a collection initializer to create the data source. Note that each element
    // in the list contains an inner sequence of scores.
    List<Student> students = new List<Student>
    {
        new Student {First="Svetlana", Last="Omelchenko", ID=111},
        new Student {First="Claire", Last="O'Donnell", ID=112},
        new Student {First="Sven", Last="Mortensen", ID=113},
        new Student {First="Cesar", Last="Garcia", ID=114},
        new Student {First="Debra", Last="Garcia", ID=115}
    };

    return students;
}
static void Main(string[] args)
{
    // Create the data source.
    List<Student> students = GetStudents();

    // Create the query.
    IEnumerable<Student> sortedStudents =
        from student in students
        orderby student.Last ascending, student.First ascending
        select student;

    // Execute the query.
    Console.WriteLine("sortedStudents:");
    foreach (Student student in sortedStudents)
        Console.WriteLine(student.Last + " " + student.First);

    // Now create groups and sort the groups. The query first sorts the names
    // of all students so that they will be in alphabetical order after they are
    // grouped. The second orderby sorts the group keys in alpha order.
    var sortedGroups =
        from student in students
        orderby student.Last, student.First
        group student by student.Last[0] into newGroup
        orderby newGroup.Key
        select newGroup;

    // Execute the query.
    Console.WriteLine(Environment.NewLine + "sortedGroups:");
    foreach (var studentGroup in sortedGroups)
    {
        Console.WriteLine(studentGroup.Key);
        foreach (var student in studentGroup)
        {
            Console.WriteLine("    {0}, {1}", student.Last, student.First);
        }
    }

    // Keep the console window open in debug mode
    Console.WriteLine("Press any key to exit.");
    Console.ReadKey();
}
}
/* Output:
sortedStudents:
Garcia Cesar
Garcia Debra
Mortensen Sven
O'Donnell Claire
Omelchenko Svetlana

sortedGroups:
G
    Garcia, Cesar
    Garcia, Debra

```

```
Garcia, Debra
M
Mortensen, Sven
O
O'Donnell, Claire
Omelchenko, Svetlana
*/
```

Remarks

At compile time, the `orderby` clause is translated to a call to the [OrderBy](#) method. Multiple keys in the `orderby` clause translate to [ThenBy](#) method calls.

See also

- [C# Reference](#)
- [Query Keywords \(LINQ\)](#)
- [LINQ in C#](#)
- [group clause](#)
- [Language Integrated Query \(LINQ\)](#)

join clause (C# Reference)

12/28/2021 • 10 minutes to read • [Edit Online](#)

The `join` clause is useful for associating elements from different source sequences that have no direct relationship in the object model. The only requirement is that the elements in each source share some value that can be compared for equality. For example, a food distributor might have a list of suppliers of a certain product, and a list of buyers. A `join` clause can be used, for example, to create a list of the suppliers and buyers of that product who are all in the same specified region.

A `join` clause takes two source sequences as input. The elements in each sequence must either be or contain a property that can be compared to a corresponding property in the other sequence. The `join` clause compares the specified keys for equality by using the special `equals` keyword. All joins performed by the `join` clause are equijoins. The shape of the output of a `join` clause depends on the specific type of join you are performing. The following are three most common join types:

- Inner join
- Group join
- Left outer join

Inner join

The following example shows a simple inner equijoin. This query produces a flat sequence of "product name / category" pairs. The same category string will appear in multiple elements. If an element from `categories` has no matching `products`, that category will not appear in the results.

```
var innerJoinQuery =  
    from category in categories  
    join prod in products on category.ID equals prod.CategoryID  
    select new { ProductName = prod.Name, Category = category.Name }; //produces flat sequence
```

For more information, see [Perform inner joins](#).

Group join

A `join` clause with an `into` expression is called a group join.

```
var innerGroupJoinQuery =  
    from category in categories  
    join prod in products on category.ID equals prod.CategoryID into prodGroup  
    select new { CategoryName = category.Name, Products = prodGroup };
```

A group join produces a hierarchical result sequence, which associates elements in the left source sequence with one or more matching elements in the right side source sequence. A group join has no equivalent in relational terms; it is essentially a sequence of object arrays.

If no elements from the right source sequence are found to match an element in the left source, the `join` clause will produce an empty array for that item. Therefore, the group join is still basically an inner-equijoin except that the result sequence is organized into groups.

If you just select the results of a group join, you can access the items, but you cannot identify the key that they

match on. Therefore, it is generally more useful to select the results of the group join into a new type that also has the key name, as shown in the previous example.

You can also, of course, use the result of a group join as the generator of another subquery:

```
var innerGroupJoinQuery2 =  
    from category in categories  
    join prod in products on category.ID equals prod.CategoryID into prodGroup  
    from prod2 in prodGroup  
    where prod2.UnitPrice > 2.50M  
    select prod2;
```

For more information, see [Perform grouped joins](#).

Left outer join

In a left outer join, all the elements in the left source sequence are returned, even if no matching elements are in the right sequence. To perform a left outer join in LINQ, use the `DefaultIfEmpty` method in combination with a group join to specify a default right-side element to produce if a left-side element has no matches. You can use `null` as the default value for any reference type, or you can specify a user-defined default type. In the following example, a user-defined default type is shown:

```
var leftOuterJoinQuery =  
    from category in categories  
    join prod in products on category.ID equals prod.CategoryID into prodGroup  
    from item in prodGroup.DefaultIfEmpty(new Product { Name = String.Empty, CategoryID = 0 })  
    select new { CatName = category.Name, ProdName = item.Name };
```

For more information, see [Perform left outer joins](#).

The equals operator

A `join` clause performs an equijoin. In other words, you can only base matches on the equality of two keys. Other types of comparisons such as "greater than" or "not equals" are not supported. To make clear that all joins are equijoins, the `join` clause uses the `equals` keyword instead of the `==` operator. The `equals` keyword can only be used in a `join` clause and it differs from the `==` operator in some important ways. When comparing strings, `equals` has an overload to compare by value and the operator `==` uses reference equality. When both sides of comparison have identical string variables, `equals` and `==` will reach the same result: true. That's because, when a program declares two or more equivalent string variables, the compiler stores all of them in the same location, see [Reference equality and string interning](#) for more information. Another important difference is the null comparison: `null equals null` is evaluated as false with `equals` operator, instead of `==` operator that evaluates it as true. Lastly, the scoping behavior is different: with `equals`, the left key consumes the outer source sequence, and the right key consumes the inner source. The outer source is only in scope on the left side of `equals` and the inner source sequence is only in scope on the right side.

Non-equijoins

You can perform non-equijoins, cross joins, and other custom join operations by using multiple `from` clauses to introduce new sequences independently into a query. For more information, see [Perform custom join operations](#).

Joins on object collections vs. relational tables

In a LINQ query expression, join operations are performed on object collections. Object collections cannot be

"joined" in exactly the same way as two relational tables. In LINQ, explicit `join` clauses are only required when two source sequences are not tied by any relationship. When working with LINQ to SQL, foreign key tables are represented in the object model as properties of the primary table. For example, in the Northwind database, the Customer table has a foreign key relationship with the Orders table. When you map the tables to the object model, the Customer class has an Orders property that contains the collection of Orders associated with that Customer. In effect, the join has already been done for you.

For more information about querying across related tables in the context of LINQ to SQL, see [How to: Map Database Relationships](#).

Composite keys

You can test for equality of multiple values by using a composite key. For more information, see [Join by using composite keys](#). Composite keys can be also used in a `group` clause.

Example

The following example compares the results of an inner join, a group join, and a left outer join on the same data sources by using the same matching keys. Some extra code is added to these examples to clarify the results in the console display.

```
class JoinDemonstration
{
    #region Data

    class Product
    {
        public string Name { get; set; }
        public int CategoryID { get; set; }
    }

    class Category
    {
        public string Name { get; set; }
        public int ID { get; set; }
    }

    // Specify the first data source.
    List<Category> categories = new List<Category>()
    {
        new Category {Name="Beverages", ID=001},
        new Category {Name="Condiments", ID=002},
        new Category {Name="Vegetables", ID=003},
        new Category {Name="Grains", ID=004},
        new Category {Name="Fruit", ID=005}
    };

    // Specify the second data source.
    List<Product> products = new List<Product>()
    {
        new Product {Name="Cola", CategoryID=001},
        new Product {Name="Tea", CategoryID=001},
        new Product {Name="Mustard", CategoryID=002},
        new Product {Name="Pickles", CategoryID=002},
        new Product {Name="Carrots", CategoryID=003},
        new Product {Name="Bok Choy", CategoryID=003},
        new Product {Name="Peaches", CategoryID=005},
        new Product {Name="Melons", CategoryID=005},
    };
    #endregion

    static void Main(string[] args)
    {

```

```

JoinDemonstration app = new JoinDemonstration();

app.InnerJoin();
app.GroupJoin();
app.GroupInnerJoin();
app.GroupJoin3();
app.LeftOuterJoin();
app.LeftOuterJoin2();

// Keep the console window open in debug mode.
Console.WriteLine("Press any key to exit.");
Console.ReadKey();
}

void InnerJoin()
{
    // Create the query that selects
    // a property from each element.
    var innerJoinQuery =
        from category in categories
        join prod in products on category.ID equals prod.CategoryID
        select new { Category = category.ID, Product = prod.Name };

    Console.WriteLine("InnerJoin:");
    // Execute the query. Access results
    // with a simple foreach statement.
    foreach (var item in innerJoinQuery)
    {
        Console.WriteLine("{0,-10}{1}", item.Product, item.Category);
    }
    Console.WriteLine("InnerJoin: {0} items in 1 group.", innerJoinQuery.Count());
    Console.WriteLine(System.Environment.NewLine);
}

void GroupJoin()
{
    // This is a demonstration query to show the output
    // of a "raw" group join. A more typical group join
    // is shown in the GroupInnerJoin method.
    var groupJoinQuery =
        from category in categories
        join prod in products on category.ID equals prod.CategoryID into prodGroup
        select prodGroup;

    // Store the count of total items (for demonstration only).
    int totalItems = 0;

    Console.WriteLine("Simple GroupJoin:");

    // A nested foreach statement is required to access group items.
    foreach (var prodGrouping in groupJoinQuery)
    {
        Console.WriteLine("Group:");
        foreach (var item in prodGrouping)
        {
            totalItems++;
            Console.WriteLine("    {0,-10}{1}", item.Name, item.CategoryID);
        }
    }
    Console.WriteLine("Unshaped GroupJoin: {0} items in {1} unnamed groups", totalItems,
groupJoinQuery.Count());
    Console.WriteLine(System.Environment.NewLine);
}

void GroupInnerJoin()
{
    var groupJoinQuery2 =
        from category in categories
        orderby category.ID

```

```

        join prod in products on category.ID equals prod.CategoryID into prodGroup
        select new
        {
            Category = category.Name,
            Products = from prod2 in prodGroup
                        orderby prod2.Name
                        select prod2
        };

//Console.WriteLine("GroupInnerJoin:");
int totalItems = 0;

Console.WriteLine("GroupInnerJoin:");
foreach (var productGroup in groupJoinQuery2)
{
    Console.WriteLine(productGroup.Category);
    foreach (var prodItem in productGroup.Products)
    {
        totalItems++;
        Console.WriteLine(" {0,-10} {1}", prodItem.Name, prodItem.CategoryID);
    }
}
Console.WriteLine("GroupInnerJoin: {0} items in {1} named groups", totalItems,
groupJoinQuery2.Count());
Console.WriteLine(System.Environment.NewLine);
}

void GroupJoin3()
{
    var groupJoinQuery3 =
        from category in categories
        join product in products on category.ID equals product.CategoryID into prodGroup
        from prod in prodGroup
        orderby prod.CategoryID
        select new { Category = prod.CategoryID, ProductName = prod.Name };

//Console.WriteLine("GroupInnerJoin:");
int totalItems = 0;

Console.WriteLine("GroupJoin3:");
foreach (var item in groupJoinQuery3)
{
    totalItems++;
    Console.WriteLine(" {0}:{1}", item.ProductName, item.Category);
}

Console.WriteLine("GroupJoin3: {0} items in 1 group", totalItems);
Console.WriteLine(System.Environment.NewLine);
}

void LeftOuterJoin()
{
    // Create the query.
    var leftOuterQuery =
        from category in categories
        join prod in products on category.ID equals prod.CategoryID into prodGroup
        select prodGroup.DefaultIfEmpty(new Product() { Name = "Nothing!", CategoryID = category.ID });

    // Store the count of total items (for demonstration only).
    int totalItems = 0;

    Console.WriteLine("Left Outer Join:");

    // A nested foreach statement is required to access group items
    foreach (var prodGrouping in leftOuterQuery)
    {
        Console.WriteLine("Group:");
        foreach (var item in prodGrouping)
    }
}

```



```

        foreach (var item in prodGrouping)
        {
            totalItems++;
            Console.WriteLine(" {0,-10}{1}", item.Name, item.CategoryID);
        }
    }
    Console.WriteLine("LeftOuterJoin: {0} items in {1} groups", totalItems, leftOuterQuery.Count());
    Console.WriteLine(System.Environment.NewLine);
}

void LeftOuterJoin2()
{
    // Create the query.
    var leftOuterQuery2 =
        from category in categories
        join prod in products on category.ID equals prod.CategoryID into prodGroup
        from item in prodGroup.DefaultIfEmpty()
        select new { Name = item == null ? "Nothing!" : item.Name, CategoryID = category.ID };

    Console.WriteLine("LeftOuterJoin2: {0} items in 1 group", leftOuterQuery2.Count());
    // Store the count of total items
    int totalItems = 0;

    Console.WriteLine("Left Outer Join 2:");

    // Groups have been flattened.
    foreach (var item in leftOuterQuery2)
    {
        totalItems++;
        Console.WriteLine("{0,-10}{1}", item.Name, item.CategoryID);
    }
    Console.WriteLine("LeftOuterJoin2: {0} items in 1 group", totalItems);
}
}
/*Output:

InnerJoin:
Cola      1
Tea       1
Mustard   2
Pickles   2
Carrots   3
Bok Choy  3
Peaches   5
Melons    5
InnerJoin: 8 items in 1 group.

Unshaped GroupJoin:
Group:
    Cola      1
    Tea       1
Group:
    Mustard   2
    Pickles   2
Group:
    Carrots   3
    Bok Choy  3
Group:
Group:
    Peaches   5
    Melons    5
Unshaped GroupJoin: 8 items in 5 unnamed groups

GroupInnerJoin:
Beverages
    Cola      1
    Tea       1

```

```

Condiments
    Mustard    2
    Pickles    2
Vegetables
    Bok Choy   3
    Carrots    3
Grains
Fruit
    Melons     5
    Peaches    5
GroupInnerJoin: 8 items in 5 named groups

```

```

GroupJoin3:
    Cola:1
    Tea:1
    Mustard:2
    Pickles:2
    Carrots:3
    Bok Choy:3
    Peaches:5
    Melons:5
GroupJoin3: 8 items in 1 group

```

```

Left Outer Join:
Group:
    Cola      1
    Tea       1
Group:
    Mustard   2
    Pickles   2
Group:
    Carrots   3
    Bok Choy  3
Group:
    Nothing!  4
Group:
    Peaches   5
    Melons    5
LeftOuterJoin: 9 items in 5 groups

```

```

LeftOuterJoin2: 9 items in 1 group
Left Outer Join 2:
Cola      1
Tea       1
Mustard   2
Pickles   2
Carrots   3
Bok Choy  3
Nothing!  4
Peaches   5
Melons    5
LeftOuterJoin2: 9 items in 1 group
Press any key to exit.
*/

```

Remarks

A `join` clause that is not followed by `into` is translated into a [Join](#) method call. A `join` clause that is followed by `into` is translated to a [GroupJoin](#) method call.

See also

- Query Keywords (LINQ)
- Language Integrated Query (LINQ)
- Join Operations
- group clause
- Perform left outer joins
- Perform inner joins
- Perform grouped joins
- Order the results of a join clause
- Join by using composite keys
- Compatible database systems for Visual Studio

let clause (C# Reference)

12/28/2021 • 2 minutes to read • [Edit Online](#)

In a query expression, it is sometimes useful to store the result of a sub-expression in order to use it in subsequent clauses. You can do this with the `let` keyword, which creates a new range variable and initializes it with the result of the expression you supply. Once initialized with a value, the range variable cannot be used to store another value. However, if the range variable holds a queryable type, it can be queried.

Example

In the following example `let` is used in two ways:

1. To create an enumerable type that can itself be queried.
2. To enable the query to call `ToLower` only one time on the range variable `word`. Without using `let`, you would have to call `ToLower` in each predicate in the `where` clause.

```

class LetSample1
{
    static void Main()
    {
        string[] strings =
        {
            "A penny saved is a penny earned.",
            "The early bird catches the worm.",
            "The pen is mightier than the sword."
        };

        // Split the sentence into an array of words
        // and select those whose first letter is a vowel.
        var earlyBirdQuery =
            from sentence in strings
            let words = sentence.Split(' ')
            from word in words
            let w = word.ToLower()
            where w[0] == 'a' || w[0] == 'e'
                || w[0] == 'i' || w[0] == 'o'
                || w[0] == 'u'
            select word;

        // Execute the query.
        foreach (var v in earlyBirdQuery)
        {
            Console.WriteLine("{0}" starts with a vowel", v);
        }

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
/* Output:
    "A" starts with a vowel
    "is" starts with a vowel
    "a" starts with a vowel
    "earned." starts with a vowel
    "early" starts with a vowel
    "is" starts with a vowel
*/

```

See also

- [C# Reference](#)
- [Query Keywords \(LINQ\)](#)
- [LINQ in C#](#)
- [Language Integrated Query \(LINQ\)](#)
- [Handle exceptions in query expressions](#)

ascending (C# Reference)

12/28/2021 • 2 minutes to read • [Edit Online](#)

The `ascending` contextual keyword is used in the [orderby clause](#) in query expressions to specify that the sort order is from smallest to largest. Because `ascending` is the default sort order, you do not have to specify it.

Example

The following example shows the use of `ascending` in an [orderby clause](#).

```
IEnumerable<string> sortAscendingQuery =  
    from vegetable in vegetables  
    orderby vegetable ascending  
    select vegetable;
```

See also

- [C# Reference](#)
- [LINQ in C#](#)
- [descending](#)

descending (C# Reference)

12/28/2021 • 2 minutes to read • [Edit Online](#)

The `descending` contextual keyword is used in the [orderby clause](#) in query expressions to specify that the sort order is from largest to smallest.

Example

The following example shows the use of `descending` in an [orderby clause](#).

```
IEnumerable<string> sortDescendingQuery =  
    from vegetable in vegetables  
    orderby vegetable descending  
    select vegetable;
```

See also

- [C# Reference](#)
- [LINQ in C#](#)
- [ascending](#)

on (C# Reference)

12/28/2021 • 2 minutes to read • [Edit Online](#)

The `on` contextual keyword is used in the [join clause](#) of a query expression to specify the join condition.

Example

The following example shows the use of `on` in a `join` clause.

```
var innerJoinQuery =  
    from category in categories  
    join prod in products on category.ID equals prod.CategoryID  
    select new { ProductName = prod.Name, Category = category.Name };
```

See also

- [C# Reference](#)
- [Language Integrated Query \(LINQ\)](#)

equals (C# Reference)

12/28/2021 • 2 minutes to read • [Edit Online](#)

The `equals` contextual keyword is used in a `join` clause in a query expression to compare the elements of two sequences. For more information, see [join clause](#).

Example

The following example shows the use of the `equals` keyword in a `join` clause.

```
var innerJoinQuery =  
    from category in categories  
    join prod in products on category.ID equals prod.CategoryID  
    select new { ProductName = prod.Name, Category = category.Name };
```

See also

- [Language Integrated Query \(LINQ\)](#)

by (C# Reference)

12/28/2021 • 2 minutes to read • [Edit Online](#)

The `by` contextual keyword is used in the `group` clause in a query expression to specify how the returned items should be grouped. For more information, see [group clause](#).

Example

The following example shows the use of the `by` contextual keyword in a `group` clause to specify that the students should be grouped according to the first letter of the last name of each student.

```
var query = from student in students
            group student by student.LastName[0];
```

See also

- [LINQ in C#](#)

in (C# Reference)

12/28/2021 • 2 minutes to read • [Edit Online](#)

The `in` keyword is used in the following contexts:

- [generic type parameters](#) in generic interfaces and delegates.
- As a [parameter modifier](#), which lets you pass an argument to a method by reference rather than by value.
- [foreach](#) statements.
- [from clauses](#) in LINQ query expressions.
- [join clauses](#) in LINQ query expressions.

See also

- [C# Keywords](#)
- [C# Reference](#)

C# operators and expressions (C# reference)

12/28/2021 • 5 minutes to read • [Edit Online](#)

C# provides a number of operators. Many of them are supported by the [built-in types](#) and allow you to perform basic operations with values of those types. Those operators include the following groups:

- [Arithmetic operators](#) that perform arithmetic operations with numeric operands
- [Comparison operators](#) that compare numeric operands
- [Boolean logical operators](#) that perform logical operations with `bool` operands
- [Bitwise and shift operators](#) that perform bitwise or shift operations with operands of the integral types
- [Equality operators](#) that check if their operands are equal or not

Typically, you can [overload](#) those operators, that is, specify the operator behavior for the operands of a user-defined type.

The simplest C# expressions are literals (for example, [integer](#) and [real](#) numbers) and names of variables. You can combine them into complex expressions by using operators. Operator [precedence](#) and [associativity](#) determine the order in which the operations in an expression are performed. You can use parentheses to change the order of evaluation imposed by operator precedence and associativity.

In the following code, examples of expressions are at the right-hand side of assignments:

```
int a, b, c;
a = 7;
b = a;
c = b++;
b = a + b * c;
c = a >= 100 ? b : c / 10;
a = (int)Math.Sqrt(b * b + c * c);

string s = "String literal";
char l = s[s.Length - 1];

var numbers = new List<int>(new[] { 1, 2, 3 });
b = numbers.FindLast(n => n > 1);
```

Typically, an expression produces a result and can be included in another expression. A `void` method call is an example of an expression that doesn't produce a result. It can be used only as a [statement](#), as the following example shows:

```
Console.WriteLine("Hello, world!");
```

Here are some other kinds of expressions that C# provides:

- [Interpolated string expressions](#) that provide convenient syntax to create formatted strings:

```
var r = 2.3;
var message = $"The area of a circle with radius {r} is {Math.PI * r * r:F3}.";
Console.WriteLine(message);
// Output:
// The area of a circle with radius 2.3 is 16.619.
```

- [Lambda expressions](#) that allow you to create anonymous functions:

```
int[] numbers = { 2, 3, 4, 5 };
var maximumSquare = numbers.Max(x => x * x);
Console.WriteLine(maximumSquare);
// Output:
// 25
```

- [Query expressions](#) that allow you to use query capabilities directly in C#:

```
var scores = new[] { 90, 97, 78, 68, 85 };
IEnumerable<int> highScoresQuery =
    from score in scores
    where score > 80
    orderby score descending
    select score;
Console.WriteLine(string.Join(" ", highScoresQuery));
// Output:
// 97 90 85
```

You can use an [expression body definition](#) to provide a concise definition for a method, constructor, property, indexer, or finalizer.

Operator precedence

In an expression with multiple operators, the operators with higher precedence are evaluated before the operators with lower precedence. In the following example, the multiplication is performed first because it has higher precedence than addition:

```
var a = 2 + 2 * 2;
Console.WriteLine(a); // output: 6
```

Use parentheses to change the order of evaluation imposed by operator precedence:

```
var a = (2 + 2) * 2;
Console.WriteLine(a); // output: 8
```

The following table lists the C# operators starting with the highest precedence to the lowest. The operators within each row have the same precedence.

OPERATORS	CATEGORY OR NAME
<code>x.y</code> , <code>f(x)</code> , <code>a[i]</code> , <code>x?.y</code> , <code>x?[y]</code> , <code>x++</code> , <code>x--</code> , <code>x!</code> , <code>new</code> , <code>typeof</code> , <code>checked</code> , <code>unchecked</code> , <code>default</code> , <code>nameof</code> , <code>delegate</code> , <code>sizeof</code> , <code>stackalloc</code> , <code>x->y</code>	Primary
<code>+x</code> , <code>-x</code> , <code>!x</code> , <code>~x</code> , <code>++x</code> , <code>--x</code> , <code>^x</code> , <code>(T)x</code> , <code>await</code> , <code>&x</code> , <code>*x</code> , <code>true</code> and <code>false</code>	Unary
<code>x..y</code>	Range
<code>switch</code> , <code>with</code>	<code>switch</code> and <code>with</code> expressions
<code>x * y</code> , <code>x / y</code> , <code>x % y</code>	Multiplicative
<code>x + y</code> , <code>x - y</code>	Additive

OPERATORS	CATEGORY OR NAME
<code>x << y, x >> y</code>	Shift
<code>x < y, x > y, x <= y, x >= y, is, as</code>	Relational and type-testing
<code>x == y, x != y</code>	Equality
<code>x & y</code>	Boolean logical AND or bitwise logical AND
<code>x ^ y</code>	Boolean logical XOR or bitwise logical XOR
<code>x y</code>	Boolean logical OR or bitwise logical OR
<code>x && y</code>	Conditional AND
<code>x y</code>	Conditional OR
<code>x ?? y</code>	Null-coalescing operator
<code>c ? t : f</code>	Conditional operator
<code>x = y, x += y, x -= y, x *= y, x /= y, x %= y, x &= y, x = y, x ^= y, x <<= y, x >>= y, x ??= y, =></code>	Assignment and lambda declaration

Operator associativity

When operators have the same precedence, associativity of the operators determines the order in which the operations are performed:

- *Left-associative* operators are evaluated in order from left to right. Except for the [assignment operators](#) and the [null-coalescing operators](#), all binary operators are left-associative. For example, `a + b - c` is evaluated as `(a + b) - c`.
- *Right-associative* operators are evaluated in order from right to left. The assignment operators, the null-coalescing operators, and the [conditional operator](#) `?:` are right-associative. For example, `x = y = z` is evaluated as `x = (y = z)`.

Use parentheses to change the order of evaluation imposed by operator associativity:

```
int a = 13 / 5 / 2;
int b = 13 / (5 / 2);
Console.WriteLine($"a = {a}, b = {b}"); // output: a = 1, b = 6
```

Operand evaluation

Unrelated to operator precedence and associativity, operands in an expression are evaluated from left to right. The following examples demonstrate the order in which operators and operands are evaluated:

EXPRESSION	ORDER OF EVALUATION
<code>a + b</code>	a, b, +

EXPRESSION	ORDER OF EVALUATION
<code>a + b * c</code>	a, b, c, *, +
<code>a / b + c * d</code>	a, b, /, c, d, *, +
<code>a / (b + c) * d</code>	a, b, c, +, /, d, *

Typically, all operator operands are evaluated. However, some operators evaluate operands conditionally. That is, the value of the leftmost operand of such an operator defines if (or which) other operands should be evaluated. These operators are the conditional logical [AND](#) (`&&`) and [OR](#) (`||`) operators, the [null-coalescing operators](#) `??` and `??=`, the [null-conditional operators](#) `?.` and `?[]`, and the [conditional operator](#) `?:`. For more information, see the description of each operator.

C# language specification

For more information, see the following sections of the [C# language specification](#):

- [Expressions](#)
- [Operators](#)

See also

- [C# reference](#)
- [Operator overloading](#)
- [Expression trees](#)

Arithmetic operators (C# reference)

12/28/2021 • 9 minutes to read • [Edit Online](#)

The following operators perform arithmetic operations with operands of numeric types:

- Unary `++` (increment), `--` (decrement), `+` (plus), and `-` (minus) operators
- Binary `*` (multiplication), `/` (division), `%` (remainder), `+` (addition), and `-` (subtraction) operators

Those operators are supported by all [integral](#) and [floating-point](#) numeric types.

In the case of integral types, those operators (except the `++` and `--` operators) are defined for the `int`, `uint`, `long`, and `ulong` types. When operands are of other integral types (`sbyte`, `byte`, `short`, `ushort`, or `char`), their values are converted to the `int` type, which is also the result type of an operation. When operands are of different integral or floating-point types, their values are converted to the closest containing type, if such a type exists. For more information, see the [Numeric promotions](#) section of the [C# language specification](#). The `++` and `--` operators are defined for all integral and floating-point numeric types and the `char` type.

Increment operator ++

The unary increment operator `++` increments its operand by 1. The operand must be a variable, a [property](#) access, or an [indexer](#) access.

The increment operator is supported in two forms: the postfix increment operator, `x++`, and the prefix increment operator, `++x`.

Postfix increment operator

The result of `x++` is the value of `x` *before* the operation, as the following example shows:

```
int i = 3;
Console.WriteLine(i); // output: 3
Console.WriteLine(i++); // output: 3
Console.WriteLine(i); // output: 4
```

Prefix increment operator

The result of `++x` is the value of `x` *after* the operation, as the following example shows:

```
double a = 1.5;
Console.WriteLine(a); // output: 1.5
Console.WriteLine(++a); // output: 2.5
Console.WriteLine(a); // output: 2.5
```

Decrement operator --

The unary decrement operator `--` decrements its operand by 1. The operand must be a variable, a [property](#) access, or an [indexer](#) access.

The decrement operator is supported in two forms: the postfix decrement operator, `x--`, and the prefix decrement operator, `--x`.

Postfix decrement operator

The result of `x--` is the value of `x` *before* the operation, as the following example shows:

```
int i = 3;
Console.WriteLine(i);    // output: 3
Console.WriteLine(i--); // output: 3
Console.WriteLine(i);    // output: 2
```

Prefix decrement operator

The result of `--x` is the value of `x` *after* the operation, as the following example shows:

```
double a = 1.5;
Console.WriteLine(a);    // output: 1.5
Console.WriteLine(--a);  // output: 0.5
Console.WriteLine(a);    // output: 0.5
```

Unary plus and minus operators

The unary `+` operator returns the value of its operand. The unary `-` operator computes the numeric negation of its operand.

```
Console.WriteLine(+4);    // output: 4

Console.WriteLine(-4);    // output: -4
Console.WriteLine(-(-4)); // output: 4

uint a = 5;
var b = -a;
Console.WriteLine(b);      // output: -5
Console.WriteLine(b.GetType()); // output: System.Int64

Console.WriteLine(-double.NaN); // output: NaN
```

The `ulong` type doesn't support the unary `-` operator.

Multiplication operator *

The multiplication operator `*` computes the product of its operands:

```
Console.WriteLine(5 * 2);    // output: 10
Console.WriteLine(0.5 * 2.5); // output: 1.25
Console.WriteLine(0.1m * 23.4m); // output: 2.34
```

The unary `*` operator is the [pointer indirection operator](#).

Division operator /

The division operator `/` divides its left-hand operand by its right-hand operand.

Integer division

For the operands of integer types, the result of the `/` operator is of an integer type and equals the quotient of the two operands rounded towards zero:

```
Console.WriteLine(13 / 5);    // output: 2
Console.WriteLine(-13 / 5);   // output: -2
Console.WriteLine(13 / -5);   // output: -2
Console.WriteLine(-13 / -5);  // output: 2
```

To obtain the quotient of the two operands as a floating-point number, use the `float`, `double`, or `decimal` type:

```
Console.WriteLine(13 / 5.0);    // output: 2.6

int a = 13;
int b = 5;
Console.WriteLine((double)a / b); // output: 2.6
```

Floating-point division

For the `float`, `double`, and `decimal` types, the result of the `/` operator is the quotient of the two operands:

```
Console.WriteLine(16.8f / 4.1f); // output: 4.097561
Console.WriteLine(16.8d / 4.1d); // output: 4.09756097560976
Console.WriteLine(16.8m / 4.1m); // output: 4.0975609756097560975609756098
```

If one of the operands is `decimal`, another operand can be neither `float` nor `double`, because neither `float` nor `double` is implicitly convertible to `decimal`. You must explicitly convert the `float` or `double` operand to the `decimal` type. For more information about conversions between numeric types, see [Built-in numeric conversions](#).

Remainder operator %

The remainder operator `%` computes the remainder after dividing its left-hand operand by its right-hand operand.

Integer remainder

For the operands of integer types, the result of `a % b` is the value produced by `a - (a / b) * b`. The sign of the non-zero remainder is the same as that of the left-hand operand, as the following example shows:

```
Console.WriteLine(5 % 4);    // output: 1
Console.WriteLine(5 % -4);   // output: 1
Console.WriteLine(-5 % 4);   // output: -1
Console.WriteLine(-5 % -4);  // output: -1
```

Use the [Math.DivRem](#) method to compute both integer division and remainder results.

Floating-point remainder

For the `float` and `double` operands, the result of `x % y` for the finite `x` and `y` is the value `z` such that

- The sign of `z`, if non-zero, is the same as the sign of `x`.
- The absolute value of `z` is the value produced by `|x| - n * |y|` where `n` is the largest possible integer that is less than or equal to `|x| / |y|` and `|x|` and `|y|` are the absolute values of `x` and `y`, respectively.

NOTE

This method of computing the remainder is analogous to that used for integer operands, but different from the IEEE 754 specification. If you need the remainder operation that complies with the IEEE 754 specification, use the [Math.IEEERemainder](#) method.

For information about the behavior of the `%` operator with non-finite operands, see the [Remainder operator](#) section of the [C# language specification](#).

For the `decimal` operands, the remainder operator `%` is equivalent to the [remainder operator](#) of the [System.Decimal](#) type.

The following example demonstrates the behavior of the remainder operator with floating-point operands:

```
Console.WriteLine(-5.2f % 2.0f); // output: -1.2
Console.WriteLine(5.9 % 3.1);    // output: 2.8
Console.WriteLine(5.9m % 3.1m); // output: 2.8
```

Addition operator +

The addition operator `+` computes the sum of its operands:

```
Console.WriteLine(5 + 4);          // output: 9
Console.WriteLine(5 + 4.3);        // output: 9.3
Console.WriteLine(5.1m + 4.2m);    // output: 9.3
```

You can also use the `+` operator for string concatenation and delegate combination. For more information, see the [+ and += operators](#) article.

Subtraction operator -

The subtraction operator `-` subtracts its right-hand operand from its left-hand operand:

```
Console.WriteLine(47 - 3);          // output: 44
Console.WriteLine(5 - 4.3);         // output: 0.7
Console.WriteLine(7.5m - 2.3m);     // output: 5.2
```

You can also use the `-` operator for delegate removal. For more information, see the [- and -= operators](#) article.

Compound assignment

For a binary operator `op`, a compound assignment expression of the form

```
x op= y
```

is equivalent to

```
x = x op y
```

except that `x` is only evaluated once.

The following example demonstrates the usage of compound assignment with arithmetic operators:

```
int a = 5;
a += 9;
Console.WriteLine(a); // output: 14

a -= 4;
Console.WriteLine(a); // output: 10

a *= 2;
Console.WriteLine(a); // output: 20

a /= 4;
Console.WriteLine(a); // output: 5

a %= 3;
Console.WriteLine(a); // output: 2
```

Because of [numeric promotions](#), the result of the `op` operation might be not implicitly convertible to the type `T` of `x`. In such a case, if `op` is a predefined operator and the result of the operation is explicitly convertible to the type `T` of `x`, a compound assignment expression of the form `x op= y` is equivalent to `x = (T)(x op y)`, except that `x` is only evaluated once. The following example demonstrates that behavior:

```
byte a = 200;
byte b = 100;

var c = a + b;
Console.WriteLine(c.GetType()); // output: System.Int32
Console.WriteLine(c); // output: 300

a += b;
Console.WriteLine(a); // output: 44
```

You also use the `+=` and `-=` operators to subscribe to and unsubscribe from an [event](#), respectively. For more information, see [How to subscribe to and unsubscribe from events](#).

Operator precedence and associativity

The following list orders arithmetic operators starting from the highest precedence to the lowest:

- Postfix increment `x++` and decrement `x--` operators
- Prefix increment `++x` and decrement `--x` and unary `+` and `-` operators
- Multiplicative `*`, `/`, and `%` operators
- Additive `+` and `-` operators

Binary arithmetic operators are left-associative. That is, operators with the same precedence level are evaluated from left to right.

Use parentheses, `()`, to change the order of evaluation imposed by operator precedence and associativity.

```
Console.WriteLine(2 + 2 * 2); // output: 6
Console.WriteLine((2 + 2) * 2); // output: 8

Console.WriteLine(9 / 5 / 2); // output: 0
Console.WriteLine(9 / (5 / 2)); // output: 4
```

For the complete list of C# operators ordered by precedence level, see the [Operator precedence](#) section of the [C# operators](#) article.

Arithmetic overflow and division by zero

When the result of an arithmetic operation is outside the range of possible finite values of the involved numeric type, the behavior of an arithmetic operator depends on the type of its operands.

Integer arithmetic overflow

Integer division by zero always throws a [DivideByZeroException](#).

In case of integer arithmetic overflow, an overflow checking context, which can be [checked](#) or [unchecked](#), controls the resulting behavior:

- In a checked context, if overflow happens in a constant expression, a compile-time error occurs. Otherwise, when the operation is performed at run time, an [OverflowException](#) is thrown.
- In an unchecked context, the result is truncated by discarding any high-order bits that don't fit in the destination type.

Along with the [checked](#) and [unchecked](#) statements, you can use the `checked` and `unchecked` operators to control the overflow checking context, in which an expression is evaluated:

```
int a = int.MaxValue;
int b = 3;

Console.WriteLine(unchecked(a + b)); // output: -2147483646
try
{
    int d = checked(a + b);
}
catch(OverflowException)
{
    Console.WriteLine($"Overflow occurred when adding {a} to {b}.");
}
```

By default, arithmetic operations occur in an *unchecked* context.

Floating-point arithmetic overflow

Arithmetic operations with the `float` and `double` types never throw an exception. The result of arithmetic operations with those types can be one of special values that represent infinity and not-a-number:

```
double a = 1.0 / 0.0;
Console.WriteLine(a); // output: Infinity
Console.WriteLine(double.IsInfinity(a)); // output: True

Console.WriteLine(double.MaxValue + double.MaxValue); // output: Infinity

double b = 0.0 / 0.0;
Console.WriteLine(b); // output: NaN
Console.WriteLine(double.IsNaN(b)); // output: True
```

For the operands of the `decimal` type, arithmetic overflow always throws an [OverflowException](#) and division by zero always throws a [DivideByZeroException](#).

Round-off errors

Because of general limitations of the floating-point representation of real numbers and floating-point arithmetic, round-off errors might occur in calculations with floating-point types. That is, the produced result of an expression might differ from the expected mathematical result. The following example demonstrates several such cases:

```
Console.WriteLine(.41f % .2f); // output: 0.00999999
```



```
double a = 0.1;  
double b = 3 * a;  
Console.WriteLine(b == 0.3);    // output: False  
Console.WriteLine(b - 0.3);     // output: 5.55111512312578E-17
```



```
decimal c = 1 / 3.0m;  
decimal d = 3 * c;  
Console.WriteLine(d == 1.0m);   // output: False  
Console.WriteLine(d);           // output: 0.999999999999999999999999999999
```

For more information, see remarks at the [System.Double](#), [System.Single](#), or [System.Decimal](#) reference pages.

Operator overloadability

A user-defined type can [overload](#) the unary (`++`, `--`, `+`, and `-`) and binary (`*`, `/`, `%`, `+`, and `-`) arithmetic operators. When a binary operator is overloaded, the corresponding compound assignment operator is also implicitly overloaded. A user-defined type cannot explicitly overload a compound assignment operator.

C# language specification

For more information, see the following sections of the [C# language specification](#):

- Postfix increment and decrement operators
- Prefix increment and decrement operators
- Unary plus operator
- Unary minus operator
- Multiplication operator
- Division operator
- Remainder operator
- Addition operator
- Subtraction operator
- Compound assignment
- The checked and unchecked operators
- Numeric promotions

See also

- C# reference
- C# operators and expressions
- System.Math
- System.MathF
- Numerics in .NET

Boolean logical operators (C# reference)

12/28/2021 • 7 minutes to read • [Edit Online](#)

The following operators perform logical operations with `bool` operands:

- Unary `!` (logical negation) operator.
- Binary `&` (logical AND), `|` (logical OR), and `^` (logical exclusive OR) operators. Those operators always evaluate both operands.
- Binary `&&` (conditional logical AND) and `||` (conditional logical OR) operators. Those operators evaluate the right-hand operand only if it's necessary.

For operands of the [integral numeric types](#), the `&`, `|`, and `^` operators perform bitwise logical operations. For more information, see [Bitwise and shift operators](#).

Logical negation operator !

The unary prefix `!` operator computes logical negation of its operand. That is, it produces `true`, if the operand evaluates to `false`, and `false`, if the operand evaluates to `true`:

```
bool passed = false;
Console.WriteLine(!passed); // output: True
Console.WriteLine(!true);   // output: False
```

Beginning with C# 8.0, the unary postfix `!` operator is the [null-forgiving operator](#).

Logical AND operator &

The `&` operator computes the logical AND of its operands. The result of `x & y` is `true` if both `x` and `y` evaluate to `true`. Otherwise, the result is `false`.

The `&` operator evaluates both operands even if the left-hand operand evaluates to `false`, so that the operation result is `false` regardless of the value of the right-hand operand.

In the following example, the right-hand operand of the `&` operator is a method call, which is performed regardless of the value of the left-hand operand:

```
bool SecondOperand()
{
    Console.WriteLine("Second operand is evaluated.");
    return true;
}

bool a = false & SecondOperand();
Console.WriteLine(a);
// Output:
// Second operand is evaluated.
// False

bool b = true & SecondOperand();
Console.WriteLine(b);
// Output:
// Second operand is evaluated.
// True
```

The [conditional logical AND operator](#) `&&` also computes the logical AND of its operands, but doesn't evaluate the right-hand operand if the left-hand operand evaluates to `false`.

For operands of the [integral numeric types](#), the `&` operator computes the [bitwise logical AND](#) of its operands. The unary `&` operator is the [address-of operator](#).

Logical exclusive OR operator `^`

The `^` operator computes the logical exclusive OR, also known as the logical XOR, of its operands. The result of `x ^ y` is `true` if `x` evaluates to `true` and `y` evaluates to `false`, or `x` evaluates to `false` and `y` evaluates to `true`. Otherwise, the result is `false`. That is, for the `bool` operands, the `^` operator computes the same result as the [inequality operator](#) `!=`.

```
Console.WriteLine(true ^ true);    // output: False
Console.WriteLine(true ^ false);   // output: True
Console.WriteLine(false ^ true);   // output: True
Console.WriteLine(false ^ false);  // output: False
```

For operands of the [integral numeric types](#), the `^` operator computes the [bitwise logical exclusive OR](#) of its operands.

Logical OR operator `|`

The `|` operator computes the logical OR of its operands. The result of `x | y` is `true` if either `x` or `y` evaluates to `true`. Otherwise, the result is `false`.

The `|` operator evaluates both operands even if the left-hand operand evaluates to `true`, so that the operation result is `true` regardless of the value of the right-hand operand.

In the following example, the right-hand operand of the `|` operator is a method call, which is performed regardless of the value of the left-hand operand:

```
bool SecondOperand()
{
    Console.WriteLine("Second operand is evaluated.");
    return true;
}

bool a = true | SecondOperand();
Console.WriteLine(a);
// Output:
// Second operand is evaluated.
// True

bool b = false | SecondOperand();
Console.WriteLine(b);
// Output:
// Second operand is evaluated.
// True
```

The [conditional logical OR operator](#) `||` also computes the logical OR of its operands, but doesn't evaluate the right-hand operand if the left-hand operand evaluates to `true`.

For operands of the [integral numeric types](#), the `|` operator computes the [bitwise logical OR](#) of its operands.

Conditional logical AND operator `&&`

The conditional logical AND operator `&&`, also known as the "short-circuiting" logical AND operator, computes the logical AND of its operands. The result of `x && y` is `true` if both `x` and `y` evaluate to `true`. Otherwise, the result is `false`. If `x` evaluates to `false`, `y` is not evaluated.

In the following example, the right-hand operand of the `&&` operator is a method call, which isn't performed if the left-hand operand evaluates to `false`:

```
bool SecondOperand()
{
    Console.WriteLine("Second operand is evaluated.");
    return true;
}

bool a = false && SecondOperand();
Console.WriteLine(a);
// Output:
// False

bool b = true && SecondOperand();
Console.WriteLine(b);
// Output:
// Second operand is evaluated.
// True
```

The [logical AND operator](#) `&` also computes the logical AND of its operands, but always evaluates both operands.

Conditional logical OR operator `||`

The conditional logical OR operator `||`, also known as the "short-circuiting" logical OR operator, computes the logical OR of its operands. The result of `x || y` is `true` if either `x` or `y` evaluates to `true`. Otherwise, the result is `false`. If `x` evaluates to `true`, `y` is not evaluated.

In the following example, the right-hand operand of the `||` operator is a method call, which isn't performed if the left-hand operand evaluates to `true`:

```
bool SecondOperand()
{
    Console.WriteLine("Second operand is evaluated.");
    return true;
}

bool a = true || SecondOperand();
Console.WriteLine(a);
// Output:
// True

bool b = false || SecondOperand();
Console.WriteLine(b);
// Output:
// Second operand is evaluated.
// True
```

The [logical OR operator](#) `|` also computes the logical OR of its operands, but always evaluates both operands.

Nullable Boolean logical operators

For `bool?` operands, the `&` ([logical AND](#)) and `|` ([logical OR](#)) operators support the three-valued logic as follows:

- The `&` operator produces `true` only if both its operands evaluate to `true`. If either `x` or `y` evaluates to `false`, `x & y` produces `false` (even if another operand evaluates to `null`). Otherwise, the result of `x & y` is `null`.
- The `|` operator produces `false` only if both its operands evaluate to `false`. If either `x` or `y` evaluates to `true`, `x | y` produces `true` (even if another operand evaluates to `null`). Otherwise, the result of `x | y` is `null`.

The following table presents that semantics:

x	y	x&y	x y
true	true	true	true
true	false	false	true
true	null	null	true
false	true	false	true
false	false	false	false
false	null	false	null
null	true	null	true
null	false	false	null
null	null	null	null

The behavior of those operators differs from the typical operator behavior with nullable value types. Typically, an operator which is defined for operands of a value type can be also used with operands of the corresponding nullable value type. Such an operator produces `null` if any of its operands evaluates to `null`. However, the `&` and `|` operators can produce non-null even if one of the operands evaluates to `null`. For more information about the operator behavior with nullable value types, see the [Lifted operators](#) section of the [Nullable value types](#) article.

You can also use the `!` and `^` operators with `bool?` operands, as the following example shows:

```
bool? test = null;
Display(!test);           // output: null
Display(test ^ false);    // output: null
Display(test ^ null);     // output: null
Display(true ^ null);     // output: null

void Display(bool? b) => Console.WriteLine(b is null ? "null" : b.Value.ToString());
```

The conditional logical operators `&&` and `||` don't support `bool?` operands.

Compound assignment

For a binary operator `op`, a compound assignment expression of the form

```
x op= y
```

is equivalent to

```
x = x op y
```

except that `x` is only evaluated once.

The `&`, `|`, and `^` operators support compound assignment, as the following example shows:

```
bool test = true;
test &= false;
Console.WriteLine(test); // output: False

test |= true;
Console.WriteLine(test); // output: True

test ^= false;
Console.WriteLine(test); // output: True
```

NOTE

The conditional logical operators `&&` and `||` don't support compound assignment.

Operator precedence

The following list orders logical operators starting from the highest precedence to the lowest:

- Logical negation operator `!`
- Logical AND operator `&`
- Logical exclusive OR operator `^`
- Logical OR operator `|`
- Conditional logical AND operator `&&`
- Conditional logical OR operator `||`

Use parentheses, `()`, to change the order of evaluation imposed by operator precedence:

```
Console.WriteLine(true | true & false); // output: True
Console.WriteLine((true | true) & false); // output: False

bool Operand(string name, bool value)
{
    Console.WriteLine($"Operand {name} is evaluated.");
    return value;
}

var byDefaultPrecedence = Operand("A", true) || Operand("B", true) && Operand("C", false);
Console.WriteLine(byDefaultPrecedence);
// Output:
// Operand A is evaluated.
// True

var changedOrder = (Operand("A", true) || Operand("B", true)) && Operand("C", false);
Console.WriteLine(changedOrder);
// Output:
// Operand A is evaluated.
// Operand C is evaluated.
// False
```

For the complete list of C# operators ordered by precedence level, see the [Operator precedence](#) section of the [C# operators](#) article.

Operator overloadability

A user-defined type can [overload](#) the `!`, `&`, `|`, and `^` operators. When a binary operator is overloaded, the corresponding compound assignment operator is also implicitly overloaded. A user-defined type cannot explicitly overload a compound assignment operator.

A user-defined type cannot overload the conditional logical operators `&&` and `||`. However, if a user-defined type overloads the [true and false operators](#) and the `&` or `|` operator in a certain way, the `&&` or `||` operation, respectively, can be evaluated for the operands of that type. For more information, see the [User-defined conditional logical operators](#) section of the [C# language specification](#).

C# language specification

For more information, see the following sections of the [C# language specification](#):

- [Logical negation operator](#)
- [Logical operators](#)
- [Conditional logical operators](#)
- [Compound assignment](#)

See also

- [C# reference](#)
- [C# operators and expressions](#)
- [Bitwise and shift operators](#)

Bitwise and shift operators (C# reference)

12/28/2021 • 7 minutes to read • [Edit Online](#)

The following operators perform bitwise or shift operations with operands of the [integral numeric types](#) or the [char](#) type:

- Unary `~` ([bitwise complement](#)) operator
- Binary `<<` ([left shift](#)) and `>>` ([right shift](#)) shift operators
- Binary `&` ([logical AND](#)), `|` ([logical OR](#)), and `^` ([logical exclusive OR](#)) operators

Those operators are defined for the `int`, `uint`, `long`, and `ulong` types. When both operands are of other integral types (`sbyte`, `byte`, `short`, `ushort`, or `char`), their values are converted to the `int` type, which is also the result type of an operation. When operands are of different integral types, their values are converted to the closest containing integral type. For more information, see the [Numeric promotions](#) section of the [C# language specification](#).

The `&`, `|`, and `^` operators are also defined for operands of the `bool` type. For more information, see [Boolean logical operators](#).

Bitwise and shift operations never cause overflow and produce the same results in [checked and unchecked](#) contexts.

Bitwise complement operator `~`

The `~` operator produces a bitwise complement of its operand by reversing each bit:

```
uint a = 0b_0000_1111_0000_1111_0000_1111_0000_1100;
uint b = ~a;
Console.WriteLine(Convert.ToString(b, toBase: 2));
// Output:
// 11110000111100001111000011110011
```

You can also use the `~` symbol to declare finalizers. For more information, see [Finalizers](#).

Left-shift operator `<<`

The `<<` operator shifts its left-hand operand left by the number of bits defined by its right-hand operand. For information about how the right-hand operand defines the shift count, see the [Shift count of the shift operators](#) section.

The left-shift operation discards the high-order bits that are outside the range of the result type and sets the low-order empty bit positions to zero, as the following example shows:

```
uint x = 0b_1100_1001_0000_0000_0000_0000_0001_0001;
Console.WriteLine($"Before: {Convert.ToString(x, toBase: 2)}");

uint y = x << 4;
Console.WriteLine($"After: {Convert.ToString(y, toBase: 2)}");
// Output:
// Before: 110010010000000000000000000010001
// After:  100100000000000000000000100010000
```

Because the shift operators are defined only for the `int`, `uint`, `long`, and `ulong` types, the result of an operation always contains at least 32 bits. If the left-hand operand is of another integral type (`sbyte`, `byte`, `short`, `ushort`, or `char`), its value is converted to the `int` type, as the following example shows:

```
byte a = 0b_1111_0001;

var b = a << 8;
Console.WriteLine(b.GetType());
Console.WriteLine($"Shifted byte: {Convert.ToString(b, toBase: 2)}");
// Output:
// System.Int32
// Shifted byte: 1111000100000000
```

Right-shift operator >>

The `>>` operator shifts its left-hand operand right by the number of bits defined by its right-hand operand. For information about how the right-hand operand defines the shift count, see the [Shift count of the shift operators](#) section.

The right-shift operation discards the low-order bits, as the following example shows:

```
uint x = 0b_1001;
Console.WriteLine($"Before: {Convert.ToString(x, toBase: 2), 4}");

uint y = x >> 2;
Console.WriteLine($"After: {Convert.ToString(y, toBase: 2), 4}");
// Output:
// Before: 1001
// After:  10
```

The high-order empty bit positions are set based on the type of the left-hand operand as follows:

- If the left-hand operand is of type `int` or `long`, the right-shift operator performs an *arithmetic* shift: the value of the most significant bit (the sign bit) of the left-hand operand is propagated to the high-order empty bit positions. That is, the high-order empty bit positions are set to zero if the left-hand operand is non-negative and set to one if it's negative.

```
int a = int.MinValue;
Console.WriteLine($"Before: {Convert.ToString(a, toBase: 2)}");

int b = a >> 3;
Console.WriteLine($"After: {Convert.ToString(b, toBase: 2)}");
// Output:
// Before: 10000000000000000000000000000000
// After:  11110000000000000000000000000000
```

- If the left-hand operand is of type `uint` or `ulong`, the right-shift operator performs a *logical* shift: the high-order empty bit positions are always set to zero.

```
uint c = 0b_1000_0000_0000_0000_0000_0000_0000_0000;
Console.WriteLine($"Before: {Convert.ToString(c, toBase: 2), 32}");

uint d = c >> 3;
Console.WriteLine($"After: {Convert.ToString(d, toBase: 2), 32}");
// Output:
// Before: 10000000000000000000000000000000
// After:  10000000000000000000000000000000
```

Logical AND operator &

The `&` operator computes the bitwise logical AND of its integral operands:

```
uint a = 0b_1111_1000;
uint b = 0b_1001_1101;
uint c = a & b;
Console.WriteLine(Convert.ToString(c, toBase: 2));
// Output:
// 10011000
```

For `bool` operands, the `&` operator computes the [logical AND](#) of its operands. The unary `&` operator is the [address-of operator](#).

Logical exclusive OR operator ^

The `^` operator computes the bitwise logical exclusive OR, also known as the bitwise logical XOR, of its integral operands:

```
uint a = 0b_1111_1000;
uint b = 0b_0001_1100;
uint c = a ^ b;
Console.WriteLine(Convert.ToString(c, toBase: 2));
// Output:
// 11100100
```

For `bool` operands, the `^` operator computes the [logical exclusive OR](#) of its operands.

Logical OR operator |

The `|` operator computes the bitwise logical OR of its integral operands:

```
uint a = 0b_1010_0000;
uint b = 0b_1001_0001;
uint c = a | b;
Console.WriteLine(Convert.ToString(c, toBase: 2));
// Output:
// 10110001
```

For `bool` operands, the `|` operator computes the [logical OR](#) of its operands.

Compound assignment

For a binary operator `op`, a compound assignment expression of the form

```
x op= y
```

is equivalent to

```
x = x op y
```

except that `x` is only evaluated once.

The following example demonstrates the usage of compound assignment with bitwise and shift operators:

```

uint INITIAL_VALUE = 0b_1111_1000;

uint a = INITIAL_VALUE;
a &= 0b_1001_1101;
Display(a); // output: 10011000

a = INITIAL_VALUE;
a |= 0b_0011_0001;
Display(a); // output: 11111001

a = INITIAL_VALUE;
a ^= 0b_1000_0000;
Display(a); // output: 1111000

a = INITIAL_VALUE;
a <<= 2;
Display(a); // output: 1111100000

a = INITIAL_VALUE;
a >>= 4;
Display(a); // output: 1111

void Display(uint x) => Console.WriteLine($"{Convert.ToString(x, toBase: 2), 8}");

```

Because of [numeric promotions](#), the result of the `op` operation might be not implicitly convertible to the type `T` of `x`. In such a case, if `op` is a predefined operator and the result of the operation is explicitly convertible to the type `T` of `x`, a compound assignment expression of the form `x op= y` is equivalent to `x = (T)(x op y)`, except that `x` is only evaluated once. The following example demonstrates that behavior:

```

byte x = 0b_1111_0001;

int b = x << 8;
Console.WriteLine($"{Convert.ToString(b, toBase: 2)}"); // output: 1111000100000000

x <<= 8;
Console.WriteLine(x); // output: 0

```

Operator precedence

The following list orders bitwise and shift operators starting from the highest precedence to the lowest:

- Bitwise complement operator `~`
- Shift operators `<<` and `>>`
- Logical AND operator `&`
- Logical exclusive OR operator `^`
- Logical OR operator `|`

Use parentheses, `()`, to change the order of evaluation imposed by operator precedence:


```

uint a = 0b_1101;
uint b = 0b_1001;
uint c = 0b_1010;

uint d1 = a | b & c;
Display(d1); // output: 1101

uint d2 = (a | b) & c;
Display(d2); // output: 1000

void Display(uint x) => Console.WriteLine($"{Convert.ToString(x, toBase: 2), 4}");

```

For the complete list of C# operators ordered by precedence level, see the [Operator precedence](#) section of the [C# operators](#) article.

Shift count of the shift operators

For the shift operators `<<` and `>>`, the type of the right-hand operand must be `int` or a type that has a [predefined implicit numeric conversion](#) to `int`.

For the `x << count` and `x >> count` expressions, the actual shift count depends on the type of `x` as follows:

- If the type of `x` is `int` or `uint`, the shift count is defined by the low-order *five* bits of the right-hand operand. That is, the shift count is computed from `count & 0x1F` (or `count & 0b_1_1111`).
- If the type of `x` is `long` or `ulong`, the shift count is defined by the low-order *six* bits of the right-hand operand. That is, the shift count is computed from `count & 0x3F` (or `count & 0b_11_1111`).

The following example demonstrates that behavior:

```

int count1 = 0b_0000_0001;
int count2 = 0b_1110_0001;

int a = 0b_0001;
Console.WriteLine($"{a} << {count1} is {a << count1}; {a} << {count2} is {a << count2}");
// Output:
// 1 << 1 is 2; 1 << 225 is 2

int b = 0b_0100;
Console.WriteLine($"{b} >> {count1} is {b >> count1}; {b} >> {count2} is {b >> count2}");
// Output:
// 4 >> 1 is 2; 4 >> 225 is 2

```

NOTE

As the preceding example shows, the result of a shift operation can be non-zero even if the value of the right-hand operand is greater than the number of bits in the left-hand operand.

Enumeration logical operators

The `~`, `&`, `|`, and `^` operators are also supported by any [enumeration](#) type. For operands of the same enumeration type, a logical operation is performed on the corresponding values of the underlying integral type. For example, for any `x` and `y` of an enumeration type `T` with an underlying type `U`, the `x & y` expression produces the same result as the `(T)((U)x & (U)y)` expression.

You typically use bitwise logical operators with an enumeration type that is defined with the [Flags](#) attribute. For more information, see the [Enumeration types as bit flags](#) section of the [Enumeration types](#) article.

Operator overloadability

A user-defined type can [overload](#) the `~`, `<<`, `>>`, `&`, `|`, and `^` operators. When a binary operator is overloaded, the corresponding compound assignment operator is also implicitly overloaded. A user-defined type cannot explicitly overload a compound assignment operator.

If a user-defined type `T` overloads the `<<` or `>>` operator, the type of the left-hand operand must be `T` and the type of the right-hand operand must be `int`.

C# language specification

For more information, see the following sections of the [C# language specification](#):

- [Bitwise complement operator](#)
- [Shift operators](#)
- [Logical operators](#)
- [Compound assignment](#)
- [Numeric promotions](#)

See also

- [C# reference](#)
- [C# operators and expressions](#)
- [Boolean logical operators](#)

Equality operators (C# reference)

12/28/2021 • 4 minutes to read • [Edit Online](#)

The `==` (equality) and `!=` (inequality) operators check if their operands are equal or not.

Equality operator `==`

The equality operator `==` returns `true` if its operands are equal, `false` otherwise.

Value types equality

Operands of the [built-in value types](#) are equal if their values are equal:

```
int a = 1 + 2 + 3;
int b = 6;
Console.WriteLine(a == b); // output: True

char c1 = 'a';
char c2 = 'A';
Console.WriteLine(c1 == c2); // output: False
Console.WriteLine(c1 == char.ToLower(c2)); // output: True
```

NOTE

For the `==`, `<`, `>`, `<=`, and `>=` operators, if any of the operands is not a number ([Double.NaN](#) or [Single.NaN](#)), the result of operation is `false`. That means that the `NaN` value is neither greater than, less than, nor equal to any other `double` (or `float`) value, including `NaN`. For more information and examples, see the [Double.NaN](#) or [Single.NaN](#) reference article.

Two operands of the same [enum](#) type are equal if the corresponding values of the underlying integral type are equal.

User-defined [struct](#) types don't support the `==` operator by default. To support the `==` operator, a user-defined struct must [overload](#) it.

Beginning with C# 7.3, the `==` and `!=` operators are supported by C# [tuples](#). For more information, see the [Tuple equality](#) section of the [Tuple types](#) article.

Reference types equality

By default, two non-record reference-type operands are equal if they refer to the same object:

```

public class ReferenceTypesEquality
{
    public class MyClass
    {
        private int id;

        public MyClass(int id) => this.id = id;
    }

    public static void Main()
    {
        var a = new MyClass(1);
        var b = new MyClass(1);
        var c = a;
        Console.WriteLine(a == b); // output: False
        Console.WriteLine(a == c); // output: True
    }
}

```

As the example shows, user-defined reference types support the `==` operator by default. However, a reference type can overload the `==` operator. If a reference type overloads the `==` operator, use the [Object.ReferenceEquals](#) method to check if two references of that type refer to the same object.

Record types equality

Available in C# 9.0 and later, [record types](#) support the `==` and `!=` operators that by default provide value equality semantics. That is, two record operands are equal when both of them are `null` or corresponding values of all fields and auto-implemented properties are equal.

```

public class RecordTypesEquality
{
    public record Point(int X, int Y, string Name);
    public record TaggedNumber(int Number, List<string> Tags);

    public static void Main()
    {
        var p1 = new Point(2, 3, "A");
        var p2 = new Point(1, 3, "B");
        var p3 = new Point(2, 3, "A");

        Console.WriteLine(p1 == p2); // output: False
        Console.WriteLine(p1 == p3); // output: True

        var n1 = new TaggedNumber(2, new List<string>() { "A" });
        var n2 = new TaggedNumber(2, new List<string>() { "A" });
        Console.WriteLine(n1 == n2); // output: False
    }
}

```

As the preceding example shows, in case of non-record reference-type members their reference values are compared, not the referenced instances.

String equality

Two [string](#) operands are equal when both of them are `null` or both string instances are of the same length and have identical characters in each character position:

```
string s1 = "hello!";
string s2 = "HeLLo!";
Console.WriteLine(s1 == s2.ToLower()); // output: True

string s3 = "Hello!";
Console.WriteLine(s1 == s3); // output: False
```

That is a case-sensitive ordinal comparison. For more information about string comparison, see [How to compare strings in C#](#).

Delegate equality

Two [delegate](#) operands of the same run-time type are equal when both of them are `null` or their invocation lists are of the same length and have equal entries in each position:

```
Action a = () => Console.WriteLine("a");

Action b = a + a;
Action c = a + a;
Console.WriteLine(object.ReferenceEquals(b, c)); // output: False
Console.WriteLine(b == c); // output: True
```

For more information, see the [Delegate equality operators](#) section of the [C# language specification](#).

Delegates that are produced from evaluation of semantically identical [lambda expressions](#) are not equal, as the following example shows:

```
Action a = () => Console.WriteLine("a");
Action b = () => Console.WriteLine("a");

Console.WriteLine(a == b); // output: False
Console.WriteLine(a + b == a + b); // output: True
Console.WriteLine(b + a == a + b); // output: False
```

Inequality operator !=

The inequality operator `!=` returns `true` if its operands are not equal, `false` otherwise. For the operands of the [built-in types](#), the expression `x != y` produces the same result as the expression `!(x == y)`. For more information about type equality, see the [Equality operator](#) section.

The following example demonstrates the usage of the `!=` operator:

```
int a = 1 + 1 + 2 + 3;
int b = 6;
Console.WriteLine(a != b); // output: True

string s1 = "Hello";
string s2 = "Hello";
Console.WriteLine(s1 != s2); // output: False

object o1 = 1;
object o2 = 1;
Console.WriteLine(o1 != o2); // output: True
```

Operator overloadability

A user-defined type can [overload](#) the `==` and `!=` operators. If a type overloads one of the two operators, it

must also overload the other one.

A record type cannot explicitly overload the `==` and `!=` operators. If you need to change the behavior of the `==` and `!=` operators for record type `T`, implement the [IEquatable<T>.Equals](#) method with the following signature:

```
public virtual bool Equals(T? other);
```

C# language specification

For more information, see the [Relational and type-testing operators](#) section of the [C# language specification](#).

For more information about equality of record types, see the [Equality members](#) section of the [records feature proposal note](#).

See also

- [C# reference](#)
- [C# operators and expressions](#)
- [System.IEquatable<T>](#)
- [Object.Equals](#)
- [Object.ReferenceEquals](#)
- [Equality comparisons](#)
- [Comparison operators](#)

Comparison operators (C# reference)

12/28/2021 • 2 minutes to read • [Edit Online](#)

The `<` (less than), `>` (greater than), `<=` (less than or equal), and `>=` (greater than or equal) comparison, also known as relational, operators compare their operands. Those operators are supported by all [integral](#) and [floating-point](#) numeric types.

NOTE

For the `==`, `<`, `>`, `<=`, and `>=` operators, if any of the operands is not a number ([Double.NaN](#) or [Single.NaN](#)), the result of operation is `false`. That means that the `NaN` value is neither greater than, less than, nor equal to any other `double` (or `float`) value, including `NaN`. For more information and examples, see the [Double.NaN](#) or [Single.NaN](#) reference article.

The [char](#) type also supports comparison operators. In the case of `char` operands, the corresponding character codes are compared.

Enumeration types also support comparison operators. For operands of the same [enum](#) type, the corresponding values of the underlying integral type are compared.

The `==` and `!=` operators check if their operands are equal or not.

Less than operator <

The `<` operator returns `true` if its left-hand operand is less than its right-hand operand, `false` otherwise:

```
Console.WriteLine(7.0 < 5.1); // output: False
Console.WriteLine(5.1 < 5.1); // output: False
Console.WriteLine(0.0 < 5.1); // output: True

Console.WriteLine(double.NaN < 5.1); // output: False
Console.WriteLine(double.NaN >= 5.1); // output: False
```

Greater than operator >

The `>` operator returns `true` if its left-hand operand is greater than its right-hand operand, `false` otherwise:

```
Console.WriteLine(7.0 > 5.1); // output: True
Console.WriteLine(5.1 > 5.1); // output: False
Console.WriteLine(0.0 > 5.1); // output: False

Console.WriteLine(double.NaN > 5.1); // output: False
Console.WriteLine(double.NaN <= 5.1); // output: False
```

Less than or equal operator <=

The `<=` operator returns `true` if its left-hand operand is less than or equal to its right-hand operand, `false` otherwise:

```
Console.WriteLine(7.0 <= 5.1);    // output: False
Console.WriteLine(5.1 <= 5.1);    // output: True
Console.WriteLine(0.0 <= 5.1);    // output: True

Console.WriteLine(double.NaN > 5.1);    // output: False
Console.WriteLine(double.NaN <= 5.1);    // output: False
```

Greater than or equal operator >=

The `>=` operator returns `true` if its left-hand operand is greater than or equal to its right-hand operand, `false` otherwise:

```
Console.WriteLine(7.0 >= 5.1);    // output: True
Console.WriteLine(5.1 >= 5.1);    // output: True
Console.WriteLine(0.0 >= 5.1);    // output: False

Console.WriteLine(double.NaN < 5.1);    // output: False
Console.WriteLine(double.NaN >= 5.1);    // output: False
```

Operator overloadability

A user-defined type can [overload](#) the `<`, `>`, `<=`, and `>=` operators.

If a type overloads one of the `<` or `>` operators, it must overload both `<` and `>`. If a type overloads one of the `<=` or `>=` operators, it must overload both `<=` and `>=`.

C# language specification

For more information, see the [Relational and type-testing operators](#) section of the [C# language specification](#).

See also

- [C# reference](#)
- [C# operators and expressions](#)
- [System.IComparable<T>](#)
- [Equality operators](#)

Member access operators and expressions (C# reference)

12/28/2021 • 8 minutes to read • [Edit Online](#)

You can use the following operators and expressions when you access a type member:

- `.` ([member access](#)): to access a member of a namespace or a type
- `[]` ([array element or indexer access](#)): to access an array element or a type indexer
- `?.` and `?[]` ([null-conditional operators](#)): to perform a member or element access operation only if an operand is non-null
- `()` ([invocation](#)): to call an accessed method or invoke a delegate
- `^` ([index from end](#)): to indicate that the element position is from the end of a sequence
- `..` ([range](#)): to specify a range of indices that you can use to obtain a range of sequence elements

Member access expression `.`

You use the `.` token to access a member of a namespace or a type, as the following examples demonstrate:

- Use `.` to access a nested namespace within a namespace, as the following example of a `using` directive shows:

```
using System.Collections.Generic;
```

- Use `.` to form a *qualified name* to access a type within a namespace, as the following code shows:

```
System.Collections.Generic.IEnumerable<int> numbers = new int[] { 1, 2, 3 };
```

Use a `using` directive to make the use of qualified names optional.

- Use `.` to access [type members](#), static and non-static, as the following code shows:

```
var constants = new List<double>();
constants.Add(Math.PI);
constants.Add(Math.E);
Console.WriteLine($"{constants.Count} values to show:");
Console.WriteLine(string.Join(", ", constants));
// Output:
// 2 values to show:
// 3.14159265358979, 2.71828182845905
```

You can also use `.` to access an [extension method](#).

Indexer operator `[]`

Square brackets, `[]`, are typically used for array, indexer, or pointer element access.

Array access

The following example demonstrates how to access array elements:

```
int[] fib = new int[10];
fib[0] = fib[1] = 1;
for (int i = 2; i < fib.Length; i++)
{
    fib[i] = fib[i - 1] + fib[i - 2];
}
Console.WriteLine(fib[fib.Length - 1]); // output: 55

double[,] matrix = new double[2,2];
matrix[0,0] = 1.0;
matrix[0,1] = 2.0;
matrix[1,0] = matrix[1,1] = 3.0;
var determinant = matrix[0,0] * matrix[1,1] - matrix[1,0] * matrix[0,1];
Console.WriteLine(determinant); // output: -3
```

If an array index is outside the bounds of the corresponding dimension of an array, an [IndexOutOfRangeException](#) is thrown.

As the preceding example shows, you also use square brackets when you declare an array type or instantiate an array instance.

For more information about arrays, see [Arrays](#).

Indexer access

The following example uses the .NET [Dictionary<TKey,TValue>](#) type to demonstrate indexer access:

```
var dict = new Dictionary<string, double>();
dict["one"] = 1;
dict["pi"] = Math.PI;
Console.WriteLine(dict["one"] + dict["pi"]); // output: 4.14159265358979
```

Indexers allow you to index instances of a user-defined type in the similar way as array indexing. Unlike array indices, which must be integer, the indexer parameters can be declared to be of any type.

For more information about indexers, see [Indexers](#).

Other usages of []

For information about pointer element access, see the [Pointer element access operator \[\]](#) section of the [Pointer related operators](#) article.

You also use square brackets to specify [attributes](#):

```
[System.Diagnostics.Conditional("DEBUG")]
void TraceMethod() {}
```

Null-conditional operators ?. and ?[]

Available in C# 6 and later, a null-conditional operator applies a [member access](#), `?.`, or [element access](#), `?[]`, operation to its operand only if that operand evaluates to non-null; otherwise, it returns `null`. That is,

- If `a` evaluates to `null`, the result of `a?.x` or `a?[x]` is `null`.
- If `a` evaluates to non-null, the result of `a?.x` or `a?[x]` is the same as the result of `a.x` or `a[x]`, respectively.

NOTE

If `a.x` or `a[x]` throws an exception, `a?.x` or `a?[x]` would throw the same exception for non-null `a`. For example, if `a` is a non-null array instance and `x` is outside the bounds of `a`, `a?[x]` would throw an [IndexOutOfRangeException](#).

The null-conditional operators are short-circuiting. That is, if one operation in a chain of conditional member or element access operations returns `null`, the rest of the chain doesn't execute. In the following example, `B` is not evaluated if `A` evaluates to `null` and `C` is not evaluated if `A` or `B` evaluates to `null`:

```
A?.B?.Do(C);
A?.B?[C];
```

If `A` might be null but `B` and `C` would not be null if `A` is not null, you only need to apply the null-conditional operator to `A`:

```
A?.B.C();
```

In the preceding example, `B` is not evaluated and `C()` is not called if `A` is null. However, if the chained member access is interrupted, for example by parentheses as in `(A?.B).C()`, short-circuiting doesn't happen.

The following examples demonstrate the usage of the `?.` and `?[]` operators:

```
double SumNumbers(List<double[]> setsOfNumbers, int indexOfSetToSum)
{
    return setsOfNumbers?[indexOfSetToSum]?.Sum() ?? double.NaN;
}

var sum1 = SumNumbers(null, 0);
Console.WriteLine(sum1); // output: NaN

var numberSets = new List<double[]>
{
    new[] { 1.0, 2.0, 3.0 },
    null
};

var sum2 = SumNumbers(numberSets, 0);
Console.WriteLine(sum2); // output: 6

var sum3 = SumNumbers(numberSets, 1);
Console.WriteLine(sum3); // output: NaN
```

```

using System;
using System.Collections.Generic;
using System.Linq;

namespace MemberAccessOperators2
{
    public static class NullConditionalShortCircuiting
    {
        public static void Main()
        {
            Person person = null;
            person?.Name.Write(); // no output: Write() is not called due to short-circuit.
            try
            {
                (person?.Name).Write();
            }
            catch (NullReferenceException)
            {
                Console.WriteLine("NullReferenceException");
            }
            // output: NullReferenceException
        }
    }

    public class Person
    {
        public FullName Name { get; set; }
    }

    public class FullName
    {
        public string FirstName { get; set; }
        public string LastName { get; set; }
        public void Write()
        {
            Console.WriteLine($"{FirstName} {LastName}");
        }
    }
}

```

The first of the preceding two examples also uses the [null-coalescing operator](#) `??` to specify an alternative expression to evaluate in case the result of a null-conditional operation is `null`.

If `a.x` or `a[x]` is of a non-nullable value type `T`, `a?.x` or `a?[x]` is of the corresponding [nullable value type](#) `T?`. If you need an expression of type `T`, apply the null-coalescing operator `??` to a null-conditional expression, as the following example shows:

```

int GetSumOfFirstTwoOrDefault(int[] numbers)
{
    if ((numbers?.Length ?? 0) < 2)
    {
        return 0;
    }
    return numbers[0] + numbers[1];
}

Console.WriteLine(GetSumOfFirstTwoOrDefault(null)); // output: 0
Console.WriteLine(GetSumOfFirstTwoOrDefault(new int[0])); // output: 0
Console.WriteLine(GetSumOfFirstTwoOrDefault(new[] { 3, 4, 5 })); // output: 7

```

In the preceding example, if you don't use the `??` operator, `numbers?.Length < 2` evaluates to `false` when `numbers` is `null`.

The null-conditional member access operator `?.` is also known as the Elvis operator.

Thread-safe delegate invocation

Use the `?.` operator to check if a delegate is non-null and invoke it in a thread-safe way (for example, when you [raise an event](#)), as the following code shows:

```
PropertyChanged?.Invoke(...)
```

That code is equivalent to the following code that you would use in C# 5 or earlier:

```
var handler = this.PropertyChanged;  
if (handler != null)  
{  
    handler(...);  
}
```

That is a thread-safe way to ensure that only a non-null `handler` is invoked. Because delegate instances are immutable, no thread can change the object referenced by the `handler` local variable. In particular, if the code executed by another thread unsubscribes from the `PropertyChanged` event and `PropertyChanged` becomes `null` before `handler` is invoked, the object referenced by `handler` remains unaffected. The `?.` operator evaluates its left-hand operand no more than once, guaranteeing that it cannot be changed to `null` after being verified as non-null.

Invocation expression ()

Use parentheses, `()`, to call a [method](#) or invoke a [delegate](#).

The following example demonstrates how to call a method, with or without arguments, and invoke a delegate:

```
Action<int> display = s => Console.WriteLine(s);  
  
var numbers = new List<int>();  
numbers.Add(10);  
numbers.Add(17);  
display(numbers.Count); // output: 2  
  
numbers.Clear();  
display(numbers.Count); // output: 0
```

You also use parentheses when you invoke a [constructor](#) with the `new` operator.

Other usages of ()

You also use parentheses to adjust the order in which to evaluate operations in an expression. For more information, see [C# operators](#).

[Cast expressions](#), which perform explicit type conversions, also use parentheses.

Index from end operator ^

Available in C# 8.0 and later, the `^` operator indicates the element position from the end of a sequence. For a sequence of length `length`, `^n` points to the element with offset `length - n` from the start of a sequence. For example, `^1` points to the last element of a sequence and `^length` points to the first element of a sequence.

```
int[] xs = new[] { 0, 10, 20, 30, 40 };
int last = xs[^1];
Console.WriteLine(last); // output: 40

var lines = new List<string> { "one", "two", "three", "four" };
string prelast = lines[^2];
Console.WriteLine(prelast); // output: three

string word = "Twenty";
Index toFirst = ^word.Length;
char first = word[toFirst];
Console.WriteLine(first); // output: T
```

As the preceding example shows, expression `^e` is of the [System.Index](#) type. In expression `^e`, the result of `e` must be implicitly convertible to `int`.

You can also use the `^` operator with the [range operator](#) to create a range of indices. For more information, see [Indices and ranges](#).

Range operator ..

Available in C# 8.0 and later, the `..` operator specifies the start and end of a range of indices as its operands. The left-hand operand is an *inclusive* start of a range. The right-hand operand is an *exclusive* end of a range. Either of operands can be an index from the start or from the end of a sequence, as the following example shows:

```
int[] numbers = new[] { 0, 10, 20, 30, 40, 50 };
int start = 1;
int amountToTake = 3;
int[] subset = numbers[start..(start + amountToTake)];
Display(subset); // output: 10 20 30

int margin = 1;
int[] inner = numbers[margin..^margin];
Display(inner); // output: 10 20 30 40

string line = "one two three";
int amountToTakeFromEnd = 5;
Range endIndices = ^amountToTakeFromEnd..^0;
string end = line[endIndices];
Console.WriteLine(end); // output: three

void Display<T>(IEnumerable<T> xs) => Console.WriteLine(string.Join(" ", xs));
```

As the preceding example shows, expression `a..b` is of the [System.Range](#) type. In expression `a..b`, the results of `a` and `b` must be implicitly convertible to `int` or [Index](#).

You can omit any of the operands of the `..` operator to obtain an open-ended range:

- `a..` is equivalent to `a..^0`
- `..b` is equivalent to `0..b`
- `..` is equivalent to `0..^0`

```
int[] numbers = new[] { 0, 10, 20, 30, 40, 50 };
int amountToDrop = numbers.Length / 2;

int[] rightHalf = numbers[amountToDrop..];
Display(rightHalf); // output: 30 40 50

int[] leftHalf = numbers[..^amountToDrop];
Display(leftHalf); // output: 0 10 20

int[] all = numbers[..];
Display(all); // output: 0 10 20 30 40 50

void Display<T>(IEnumerable<T> xs) => Console.WriteLine(string.Join(" ", xs));
```

For more information, see [Indices and ranges](#).

Operator overloadability

The `.`, `()`, `^`, and `..` operators cannot be overloaded. The `[]` operator is also considered a non-overloadable operator. Use [indexers](#) to support indexing with user-defined types.

C# language specification

For more information, see the following sections of the [C# language specification](#):

- [Member access](#)
- [Element access](#)
- [Null-conditional operator](#)
- [Invocation expressions](#)

For more information about indices and ranges, see the [feature proposal note](#).

See also

- [C# reference](#)
- [C# operators and expressions](#)
- [?? \(null-coalescing operator\)](#)
- [:: operator](#)

Type-testing operators and cast expression (C# reference)

12/28/2021 • 6 minutes to read • [Edit Online](#)

You can use the following operators and expressions to perform type checking or type conversion:

- **is operator**: Check if the run-time type of an expression is compatible with a given type
- **as operator**: Explicitly convert an expression to a given type if its run-time type is compatible with that type
- **cast expression**: Perform an explicit conversion
- **typeof operator**: Obtain the [System.Type](#) instance for a type

is operator

The `is` operator checks if the run-time type of an expression result is compatible with a given type. Beginning with C# 7.0, the `is` operator also tests an expression result against a pattern.

The expression with the type-testing `is` operator has the following form

```
E is T
```

where `E` is an expression that returns a value and `T` is the name of a type or a type parameter. `E` can't be an anonymous method or a lambda expression.

The `is` operator returns `true` when an expression result is non-null and any of the following conditions are true:

- The run-time type of an expression result is `T`.
- The run-time type of an expression result derives from type `T`, implements interface `T`, or another [implicit reference conversion](#) exists from it to `T`.
- The run-time type of an expression result is a [nullable value type](#) with the underlying type `T` and the `Nullable<T>.HasValue` is `true`.
- A [boxing](#) or [unboxing](#) conversion exists from the run-time type of an expression result to type `T`.

The `is` operator doesn't consider user-defined conversions.

The following example demonstrates that the `is` operator returns `true` if the run-time type of an expression result derives from a given type, that is, there exists a reference conversion between types:


```

public class Base { }

public class Derived : Base { }

public static class IsOperatorExample
{
    public static void Main()
    {
        object b = new Base();
        Console.WriteLine(b is Base); // output: True
        Console.WriteLine(b is Derived); // output: False

        object d = new Derived();
        Console.WriteLine(d is Base); // output: True
        Console.WriteLine(d is Derived); // output: True
    }
}

```

The next example shows that the `is` operator takes into account boxing and unboxing conversions but doesn't consider [numeric conversions](#):

```

int i = 27;
Console.WriteLine(i is System.IFormattable); // output: True

object iBoxed = i;
Console.WriteLine(iBoxed is int); // output: True
Console.WriteLine(iBoxed is long); // output: False

```

For information about C# conversions, see the [Conversions](#) chapter of the [C# language specification](#).

Type testing with pattern matching

Beginning with C# 7.0, the `is` operator also tests an expression result against a pattern. The following example shows how to use a [declaration pattern](#) to check the run-time type of an expression:

```

int i = 23;
object iBoxed = i;
int? jNullable = 7;
if (iBoxed is int a && jNullable is int b)
{
    Console.WriteLine(a + b); // output 30
}

```

For information about the supported patterns, see [Patterns](#).

as operator

The `as` operator explicitly converts the result of an expression to a given reference or nullable value type. If the conversion isn't possible, the `as` operator returns `null`. Unlike a [cast expression](#), the `as` operator never throws an exception.

The expression of the form

```
E as T
```

where `E` is an expression that returns a value and `T` is the name of a type or a type parameter, produces the same result as

```
E is T ? (T)(E) : (T)null
```

except that `E` is only evaluated once.

The `as` operator considers only reference, nullable, boxing, and unboxing conversions. You can't use the `as` operator to perform a user-defined conversion. To do that, use a [cast expression](#).

The following example demonstrates the usage of the `as` operator:

```
IEnumerable<int> numbers = new[] { 10, 20, 30 };
IList<int> indexable = numbers as IList<int>;
if (indexable != null)
{
    Console.WriteLine(indexable[0] + indexable[indexable.Count - 1]); // output: 40
}
```

NOTE

As the preceding example shows, you need to compare the result of the `as` expression with `null` to check if the conversion is successful. Beginning with C# 7.0, you can use the [is operator](#) both to test if the conversion succeeds and, if it succeeds, assign its result to a new variable.

Cast expression

A cast expression of the form `(T)E` performs an explicit conversion of the result of expression `E` to type `T`. If no explicit conversion exists from the type of `E` to type `T`, a compile-time error occurs. At run time, an explicit conversion might not succeed and a cast expression might throw an exception.

The following example demonstrates explicit numeric and reference conversions:

```
double x = 1234.7;
int a = (int)x;
Console.WriteLine(a); // output: 1234

IEnumerable<int> numbers = new int[] { 10, 20, 30 };
IList<int> list = (IList<int>)numbers;
Console.WriteLine(list.Count); // output: 3
Console.WriteLine(list[1]); // output: 20
```

For information about supported explicit conversions, see the [Explicit conversions](#) section of the [C# language specification](#). For information about how to define a custom explicit or implicit type conversion, see [User-defined conversion operators](#).

Other usages of ()

You also use parentheses to [call a method or invoke a delegate](#).

Other use of parentheses is to adjust the order in which to evaluate operations in an expression. For more information, see [C# operators](#).

typeof operator

The `typeof` operator obtains the [System.Type](#) instance for a type. The argument to the `typeof` operator must be the name of a type or a type parameter, as the following example shows:

```

void PrintType<T>() => Console.WriteLine(typeof(T));

Console.WriteLine(typeof(List<string>));
PrintType<int>();
PrintType<System.Int32>();
PrintType<Dictionary<int, char>>();
// Output:
// System.Collections.Generic.List`1[System.String]
// System.Int32
// System.Int32
// System.Collections.Generic.Dictionary`2[System.Int32,System.Char]

```

The argument mustn't be a type that requires metadata annotations. Examples include the following types:

- `dynamic`
- `string?` (or any nullable reference type)

These types aren't directly represented in metadata. The types include attributes that describe the underlying type. In both cases, you can use the underlying type. Instead of `dynamic`, you can use `object`. Instead of `string?`, you can use `string`.

You can also use the `typeof` operator with unbound generic types. The name of an unbound generic type must contain the appropriate number of commas, which is one less than the number of type parameters. The following example shows the usage of the `typeof` operator with an unbound generic type:

```

Console.WriteLine(typeof(Dictionary<, >));
// Output:
// System.Collections.Generic.Dictionary`2[TKey,TValue]

```

An expression can't be an argument of the `typeof` operator. To get the [System.Type](#) instance for the run-time type of an expression result, use the [Object.GetType](#) method.

Type testing with the `typeof` operator

Use the `typeof` operator to check if the run-time type of the expression result exactly matches a given type. The following example demonstrates the difference between type checking done with the `typeof` operator and the [is operator](#):

```

public class Animal { }

public class Giraffe : Animal { }

public static class TypeOfExample
{
    public static void Main()
    {
        object b = new Giraffe();
        Console.WriteLine(b is Animal); // output: True
        Console.WriteLine(b.GetType() == typeof(Animal)); // output: False

        Console.WriteLine(b is Giraffe); // output: True
        Console.WriteLine(b.GetType() == typeof(Giraffe)); // output: True
    }
}

```

Operator overloadability

The `is`, `as`, and `typeof` operators cannot be overloaded.

A user-defined type can't overload the `()` operator, but can define custom type conversions that can be performed by a cast expression. For more information, see [User-defined conversion operators](#).

C# language specification

For more information, see the following sections of the [C# language specification](#):

- [The is operator](#)
- [The as operator](#)
- [Cast expressions](#)
- [The typeof operator](#)

See also

- [C# reference](#)
- [C# operators and expressions](#)
- [How to safely cast by using pattern matching and the is and as operators](#)
- [Generics in .NET](#)

User-defined conversion operators (C# reference)

12/28/2021 • 2 minutes to read • [Edit Online](#)

A user-defined type can define a custom implicit or explicit conversion from or to another type.

Implicit conversions don't require special syntax to be invoked and can occur in a variety of situations, for example, in assignments and methods invocations. Predefined C# implicit conversions always succeed and never throw an exception. User-defined implicit conversions should behave in that way as well. If a custom conversion can throw an exception or lose information, define it as an explicit conversion.

User-defined conversions are not considered by the `is` and `as` operators. Use a [cast expression](#) to invoke a user-defined explicit conversion.

Use the `operator` and `implicit` or `explicit` keywords to define an implicit or explicit conversion, respectively. The type that defines a conversion must be either a source type or a target type of that conversion. A conversion between two user-defined types can be defined in either of the two types.

The following example demonstrates how to define an implicit and explicit conversion:

```
using System;

public readonly struct Digit
{
    private readonly byte digit;

    public Digit(byte digit)
    {
        if (digit > 9)
        {
            throw new ArgumentOutOfRangeException(nameof(digit), "Digit cannot be greater than nine.");
        }
        this.digit = digit;
    }

    public static implicit operator byte(Digit d) => d.digit;
    public static explicit operator Digit(byte b) => new Digit(b);

    public override string ToString() => $"{digit}";
}

public static class UserDefinedConversions
{
    public static void Main()
    {
        var d = new Digit(7);

        byte number = d;
        Console.WriteLine(number); // output: 7

        Digit digit = (Digit)number;
        Console.WriteLine(digit); // output: 7
    }
}
```

You also use the `operator` keyword to overload a predefined C# operator. For more information, see [Operator overloading](#).

C# language specification

For more information, see the following sections of the [C# language specification](#):

- [Conversion operators](#)
- [User-defined conversions](#)
- [Implicit conversions](#)
- [Explicit conversions](#)

See also

- [C# reference](#)
- [C# operators and expressions](#)
- [Operator overloading](#)
- [Type-testing and cast operators](#)
- [Casting and type conversion](#)
- [Design guidelines - Conversion operators](#)
- [Chained user-defined explicit conversions in C#](#)

Pointer related operators (C# reference)

12/28/2021 • 7 minutes to read • [Edit Online](#)

You can use the following operators to work with pointers:

- Unary `&` ([address-of](#)) operator: to get the address of a variable
- Unary `*` ([pointer indirection](#)) operator: to obtain the variable pointed by a pointer
- The `->` ([member access](#)) and `[]` ([element access](#)) operators
- Arithmetic operators `+`, `-`, `++`, and `--`
- Comparison operators `==`, `!=`, `<`, `>`, `<=`, and `>=`

For information about pointer types, see [Pointer types](#).

NOTE

Any operation with pointers requires an [unsafe](#) context. The code that contains unsafe blocks must be compiled with the [AllowUnsafeBlocks](#) compiler option.

Address-of operator &

The unary `&` operator returns the address of its operand:

```
unsafe
{
    int number = 27;
    int* pointerToNumber = &number;

    Console.WriteLine($"Value of the variable: {number}");
    Console.WriteLine($"Address of the variable: {(long)pointerToNumber:X}");
}
// Output is similar to:
// Value of the variable: 27
// Address of the variable: 6C1457DBD4
```

The operand of the `&` operator must be a fixed variable. *Fixed* variables are variables that reside in storage locations that are unaffected by operation of the [garbage collector](#). In the preceding example, the local variable `number` is a fixed variable, because it resides on the stack. Variables that reside in storage locations that can be affected by the garbage collector (for example, relocated) are called *movable* variables. Object fields and array elements are examples of movable variables. You can get the address of a movable variable if you "fix", or "pin", it with a [fixed statement](#). The obtained address is valid only inside the block of a `fixed` statement. The following example shows how to use a `fixed` statement and the `&` operator:

```
unsafe
{
    byte[] bytes = { 1, 2, 3 };
    fixed (byte* pointerToFirst = &bytes[0])
    {
        // The address stored in pointerToFirst
        // is valid only inside this fixed statement block.
    }
}
```

You can't get the address of a constant or a value.

For more information about fixed and movable variables, see the [Fixed and moveable variables](#) section of the [C# language specification](#).

The binary `&` operator computes the [logical AND](#) of its Boolean operands or the [bitwise logical AND](#) of its integral operands.

Pointer indirection operator *

The unary pointer indirection operator `*` obtains the variable to which its operand points. It's also known as the dereference operator. The operand of the `*` operator must be of a pointer type.

```
unsafe
{
    char letter = 'A';
    char* pointerToLetter = &letter;
    Console.WriteLine($"Value of the `letter` variable: {letter}");
    Console.WriteLine($"Address of the `letter` variable: {(long)pointerToLetter:X}");

    *pointerToLetter = 'Z';
    Console.WriteLine($"Value of the `letter` variable after update: {letter}");
}
// Output is similar to:
// Value of the `letter` variable: A
// Address of the `letter` variable: DCB977DDF4
// Value of the `letter` variable after update: Z
```

You cannot apply the `*` operator to an expression of type `void*`.

The binary `*` operator computes the [product](#) of its numeric operands.

Pointer member access operator ->

The `->` operator combines [pointer indirection](#) and [member access](#). That is, if `x` is a pointer of type `T*` and `y` is an accessible member of type `T`, an expression of the form

```
x->y
```

is equivalent to

```
(*x).y
```

The following example demonstrates the usage of the `->` operator:


```

public struct Coords
{
    public int X;
    public int Y;
    public override string ToString() => $"({X}, {Y})";
}

public class PointerMemberAccessExample
{
    public static unsafe void Main()
    {
        Coords coords;
        Coords* p = &coords;
        p->X = 3;
        p->Y = 4;
        Console.WriteLine(p->ToString()); // output: (3, 4)
    }
}

```

You cannot apply the `->` operator to an expression of type `void*`.

Pointer element access operator `[]`

For an expression `p` of a pointer type, a pointer element access of the form `p[n]` is evaluated as `*(p + n)`, where `n` must be of a type implicitly convertible to `int`, `uint`, `long`, or `ulong`. For information about the behavior of the `+` operator with pointers, see the [Addition or subtraction of an integral value to or from a pointer](#) section.

The following example demonstrates how to access array elements with a pointer and the `[]` operator:

```

unsafe
{
    char* pointerToChars = stackalloc char[123];

    for (int i = 65; i < 123; i++)
    {
        pointerToChars[i] = (char)i;
    }

    Console.Write("Uppercase letters: ");
    for (int i = 65; i < 91; i++)
    {
        Console.Write(pointerToChars[i]);
    }
}
// Output:
// Uppercase letters: ABCDEFGHIJKLMNOPQRSTUVWXYZ

```

In the preceding example, a `stackalloc` expression allocates a block of memory on the stack.

NOTE

The pointer element access operator doesn't check for out-of-bounds errors.

You cannot use `[]` for pointer element access with an expression of type `void*`.

You can also use the `[]` operator for [array element or indexer access](#).

Pointer arithmetic operators

You can perform the following arithmetic operations with pointers:

- Add or subtract an integral value to or from a pointer
- Subtract two pointers
- Increment or decrement a pointer

You cannot perform those operations with pointers of type `void*`.

For information about supported arithmetic operations with numeric types, see [Arithmetic operators](#).

Addition or subtraction of an integral value to or from a pointer

For a pointer `p` of type `T*` and an expression `n` of a type implicitly convertible to `int`, `uint`, `long`, or `ulong`, addition and subtraction are defined as follows:

- Both `p + n` and `n + p` expressions produce a pointer of type `T*` that results from adding `n * sizeof(T)` to the address given by `p`.
- The `p - n` expression produces a pointer of type `T*` that results from subtracting `n * sizeof(T)` from the address given by `p`.

The `sizeof` operator obtains the size of a type in bytes.

The following example demonstrates the usage of the `+` operator with a pointer:

```
unsafe
{
    const int Count = 3;
    int[] numbers = new int[Count] { 10, 20, 30 };
    fixed (int* pointerToFirst = &numbers[0])
    {
        int* pointerToLast = pointerToFirst + (Count - 1);

        Console.WriteLine($"Value {pointerToFirst} at address {(long)pointerToFirst}");
        Console.WriteLine($"Value {pointerToLast} at address {(long)pointerToLast}");
    }
}
// Output is similar to:
// Value 10 at address 1818345918136
// Value 30 at address 1818345918144
```

Pointer subtraction

For two pointers `p1` and `p2` of type `T*`, the expression `p1 - p2` produces the difference between the addresses given by `p1` and `p2` divided by `sizeof(T)`. The type of the result is `long`. That is, `p1 - p2` is computed as `((long)(p1) - (long)(p2)) / sizeof(T)`.

The following example demonstrates the pointer subtraction:

```
unsafe
{
    int* numbers = stackalloc int[] { 0, 1, 2, 3, 4, 5 };
    int* p1 = &numbers[1];
    int* p2 = &numbers[5];
    Console.WriteLine(p2 - p1); // output: 4
}
```

Pointer increment and decrement

The `++` increment operator [adds](#) 1 to its pointer operand. The `--` decrement operator [subtracts](#) 1 from its pointer operand.

Both operators are supported in two forms: postfix (`p++` and `p--`) and prefix (`++p` and `--p`). The result of `p++` and `p--` is the value of `p` *before* the operation. The result of `++p` and `--p` is the value of `p` *after* the operation.

The following example demonstrates the behavior of both postfix and prefix increment operators:

```
unsafe
{
    int* numbers = stackalloc int[] { 0, 1, 2 };
    int* p1 = &numbers[0];
    int* p2 = p1;
    Console.WriteLine($"Before operation: p1 - {(long)p1}, p2 - {(long)p2}");
    Console.WriteLine($"Postfix increment of p1: {(long)(p1++)}");
    Console.WriteLine($"Prefix increment of p2: {(long)(++p2)}");
    Console.WriteLine($"After operation: p1 - {(long)p1}, p2 - {(long)p2}");
}
// Output is similar to
// Before operation: p1 - 816489946512, p2 - 816489946512
// Postfix increment of p1: 816489946512
// Prefix increment of p2: 816489946516
// After operation: p1 - 816489946516, p2 - 816489946516
```

Pointer comparison operators

You can use the `==`, `!=`, `<`, `>`, `<=`, and `>=` operators to compare operands of any pointer type, including `void*`. Those operators compare the addresses given by the two operands as if they were unsigned integers.

For information about the behavior of those operators for operands of other types, see the [Equality operators](#) and [Comparison operators](#) articles.

Operator precedence

The following list orders pointer related operators starting from the highest precedence to the lowest:

- Postfix increment `x++` and decrement `x--` operators and the `->` and `[]` operators
- Prefix increment `++x` and decrement `--x` operators and the `&` and `*` operators
- Additive `+` and `-` operators
- Comparison `<`, `>`, `<=`, and `>=` operators
- Equality `==` and `!=` operators

Use parentheses, `()`, to change the order of evaluation imposed by operator precedence.

For the complete list of C# operators ordered by precedence level, see the [Operator precedence](#) section of the [C# operators](#) article.

Operator overloadability

A user-defined type cannot overload the pointer related operators `&`, `*`, `->`, and `[]`.

C# language specification

For more information, see the following sections of the [C# language specification](#):

- [Fixed and moveable variables](#)
- [The address-of operator](#)
- [Pointer indirection](#)

- [Pointer member access](#)
- [Pointer element access](#)
- [Pointer arithmetic](#)
- [Pointer increment and decrement](#)
- [Pointer comparison](#)

See also

- [C# reference](#)
- [C# operators and expressions](#)
- [Pointer types](#)
- [unsafe keyword](#)
- [fixed keyword](#)
- [stackalloc](#)
- [sizeof operator](#)

Assignment operators (C# reference)

12/28/2021 • 2 minutes to read • [Edit Online](#)

The assignment operator `=` assigns the value of its right-hand operand to a variable, a [property](#), or an [indexer](#) element given by its left-hand operand. The result of an assignment expression is the value assigned to the left-hand operand. The type of the right-hand operand must be the same as the type of the left-hand operand or implicitly convertible to it.

The assignment operator `=` is right-associative, that is, an expression of the form

```
a = b = c
```

is evaluated as

```
a = (b = c)
```

The following example demonstrates the usage of the assignment operator with a local variable, a property, and an indexer element as its left-hand operand:

```
var numbers = new List<double>() { 1.0, 2.0, 3.0 };

Console.WriteLine(numbers.Capacity);
numbers.Capacity = 100;
Console.WriteLine(numbers.Capacity);
// Output:
// 4
// 100

int newFirstElement;
double originalFirstElement = numbers[0];
newFirstElement = 5;
numbers[0] = newFirstElement;
Console.WriteLine(originalFirstElement);
Console.WriteLine(numbers[0]);
// Output:
// 1
// 5
```

ref assignment operator

Beginning with C# 7.3, you can use the ref assignment operator `= ref` to reassign a [ref local](#) or [ref readonly local](#) variable. The following example demonstrates the usage of the ref assignment operator:

```

void Display(double[] s) => Console.WriteLine(string.Join(" ", s));

double[] arr = { 0.0, 0.0, 0.0 };
Display(arr);

ref double arrayElement = ref arr[0];
arrayElement = 3.0;
Display(arr);

arrayElement = ref arr[arr.Length - 1];
arrayElement = 5.0;
Display(arr);
// Output:
// 0 0 0
// 3 0 0
// 3 0 5

```

In the case of the ref assignment operator, both of its operands must be of the same type.

Compound assignment

For a binary operator `op`, a compound assignment expression of the form

```
x op= y
```

is equivalent to

```
x = x op y
```

except that `x` is only evaluated once.

Compound assignment is supported by [arithmetic](#), [Boolean logical](#), and [bitwise logical and shift](#) operators.

Null-coalescing assignment

Beginning with C# 8.0, you can use the null-coalescing assignment operator `??=` to assign the value of its right-hand operand to its left-hand operand only if the left-hand operand evaluates to `null`. For more information, see the [?? and ??= operators](#) article.

Operator overloadability

A user-defined type cannot [overload](#) the assignment operator. However, a user-defined type can define an implicit conversion to another type. That way, the value of a user-defined type can be assigned to a variable, a property, or an indexer element of another type. For more information, see [User-defined conversion operators](#).

A user-defined type cannot explicitly overload a compound assignment operator. However, if a user-defined type overloads a binary operator `op`, the `op=` operator, if it exists, is also implicitly overloaded.

C# language specification

For more information, see the [Assignment operators](#) section of the [C# language specification](#).

For more information about the ref assignment operator `= ref`, see the [feature proposal note](#).

See also

- [C# reference](#)
- [C# operators and expressions](#)
- [ref keyword](#)

Lambda expressions (C# reference)

12/28/2021 • 15 minutes to read • [Edit Online](#)

You use a *lambda expression* to create an anonymous function. Use the [lambda declaration operator](#) `=>` to separate the lambda's parameter list from its body. A lambda expression can be of any of the following two forms:

- [Expression lambda](#) that has an expression as its body:

```
(input-parameters) => expression
```

- [Statement lambda](#) that has a statement block as its body:

```
(input-parameters) => { <sequence-of-statements> }
```

To create a lambda expression, you specify input parameters (if any) on the left side of the lambda operator and an expression or a statement block on the other side.

Any lambda expression can be converted to a [delegate](#) type. The delegate type to which a lambda expression can be converted is defined by the types of its parameters and return value. If a lambda expression doesn't return a value, it can be converted to one of the `Action` delegate types; otherwise, it can be converted to one of the `Func` delegate types. For example, a lambda expression that has two parameters and returns no value can be converted to an `Action<T1,T2>` delegate. A lambda expression that has one parameter and returns a value can be converted to a `Func<T,TResult>` delegate. In the following example, the lambda expression `x => x * x`, which specifies a parameter that's named `x` and returns the value of `x` squared, is assigned to a variable of a delegate type:

```
Func<int, int> square = x => x * x;  
Console.WriteLine(square(5));  
// Output:  
// 25
```

Expression lambdas can also be converted to the [expression tree](#) types, as the following example shows:

```
System.Linq.Expressions.Expression<Func<int, int>> e = x => x * x;  
Console.WriteLine(e);  
// Output:  
// x => (x * x)
```

You can use lambda expressions in any code that requires instances of delegate types or expression trees, for example as an argument to the [Task.Run\(Action\)](#) method to pass the code that should be executed in the background. You can also use lambda expressions when you write [LINQ in C#](#), as the following example shows:

```
int[] numbers = { 2, 3, 4, 5 };  
var squaredNumbers = numbers.Select(x => x * x);  
Console.WriteLine(string.Join(" ", squaredNumbers));  
// Output:  
// 4 9 16 25
```


When you use method-based syntax to call the [Enumerable.Select](#) method in the [System.Linq.Enumerable](#) class, for example in LINQ to Objects and LINQ to XML, the parameter is a delegate type [System.Func<T, TResult>](#). When you call the [Queryable.Select](#) method in the [System.Linq.Queryable](#) class, for example in LINQ to SQL, the parameter type is an expression tree type [Expression<Func<TSource, TResult>>](#). In both cases, you can use the same lambda expression to specify the parameter value. That makes the two `Select` calls to look similar although in fact the type of objects created from the lambdas is different.

Expression lambdas

A lambda expression with an expression on the right side of the `=>` operator is called an *expression lambda*. An expression lambda returns the result of the expression and takes the following basic form:

```
(input-parameters) => expression
```

The body of an expression lambda can consist of a method call. However, if you're creating [expression trees](#) that are evaluated outside the context of the .NET Common Language Runtime (CLR), such as in SQL Server, you shouldn't use method calls in lambda expressions. The methods will have no meaning outside the context of the .NET Common Language Runtime (CLR).

Statement lambdas

A statement lambda resembles an expression lambda except that its statements are enclosed in braces:

```
(input-parameters) => { <sequence-of-statements> }
```

The body of a statement lambda can consist of any number of statements; however, in practice there are typically no more than two or three.

```
Action<string> greet = name =>
{
    string greeting = $"Hello {name}!";
    Console.WriteLine(greeting);
};
greet("World");
// Output:
// Hello World!
```

You can't use statement lambdas to create expression trees.

Input parameters of a lambda expression

You enclose input parameters of a lambda expression in parentheses. Specify zero input parameters with empty parentheses:

```
Action line = () => Console.WriteLine();
```

If a lambda expression has only one input parameter, parentheses are optional:

```
Func<double, double> cube = x => x * x * x;
```

Two or more input parameters are separated by commas:

```
Func<int, int, bool> testForEquality = (x, y) => x == y;
```

Sometimes the compiler can't infer the types of input parameters. You can specify the types explicitly as shown in the following example:

```
Func<int, string, bool> isTooLong = (int x, string s) => s.Length > x;
```

Input parameter types must be all explicit or all implicit; otherwise, a [CS0748](#) compiler error occurs.

Beginning with C# 9.0, you can use [discards](#) to specify two or more input parameters of a lambda expression that aren't used in the expression:

```
Func<int, int, int> constant = (_, _) => 42;
```

Lambda discard parameters may be useful when you use a lambda expression to [provide an event handler](#).

NOTE

For backwards compatibility, if only a single input parameter is named `_`, then, within a lambda expression, `_` is treated as the name of that parameter.

Async lambdas

You can easily create lambda expressions and statements that incorporate asynchronous processing by using the [async](#) and [await](#) keywords. For example, the following Windows Forms example contains an event handler that calls and awaits an async method, `ExampleMethodAsync`.

```
public partial class Form1 : Form
{
    public Form1()
    {
        InitializeComponent();
        button1.Click += button1_Click;
    }

    private async void button1_Click(object sender, EventArgs e)
    {
        await ExampleMethodAsync();
        textBox1.Text += "\r\nControl returned to Click event handler.\n";
    }

    private async Task ExampleMethodAsync()
    {
        // The following line simulates a task-returning asynchronous process.
        await Task.Delay(1000);
    }
}
```

You can add the same event handler by using an async lambda. To add this handler, add an `async` modifier before the lambda parameter list, as the following example shows:

```

public partial class Form1 : Form
{
    public Form1()
    {
        InitializeComponent();
        button1.Click += async (sender, e) =>
        {
            await ExampleMethodAsync();
            textBox1.Text += "\r\nControl returned to Click event handler.\n";
        };
    }

    private async Task ExampleMethodAsync()
    {
        // The following line simulates a task-returning asynchronous process.
        await Task.Delay(1000);
    }
}

```

For more information about how to create and use async methods, see [Asynchronous Programming with async and await](#).

Lambda expressions and tuples

Starting with C# 7.0, the C# language provides built-in support for [tuples](#). You can provide a tuple as an argument to a lambda expression, and your lambda expression can also return a tuple. In some cases, the C# compiler uses type inference to determine the types of tuple components.

You define a tuple by enclosing a comma-delimited list of its components in parentheses. The following example uses tuple with three components to pass a sequence of numbers to a lambda expression, which doubles each value and returns a tuple with three components that contains the result of the multiplications.

```

Func<(int, int, int), (int, int, int)> doubleThem = ns => (2 * ns.Item1, 2 * ns.Item2, 2 * ns.Item3);
var numbers = (2, 3, 4);
var doubledNumbers = doubleThem(numbers);
Console.WriteLine($"The set {numbers} doubled: {doubledNumbers}");
// Output:
// The set (2, 3, 4) doubled: (4, 6, 8)

```

Ordinarily, the fields of a tuple are named `Item1`, `Item2`, and so on. You can, however, define a tuple with named components, as the following example does.

```

Func<(int n1, int n2, int n3), (int, int, int)> doubleThem = ns => (2 * ns.n1, 2 * ns.n2, 2 * ns.n3);
var numbers = (2, 3, 4);
var doubledNumbers = doubleThem(numbers);
Console.WriteLine($"The set {numbers} doubled: {doubledNumbers}");

```

For more information about C# tuples, see [Tuple types](#).

Lambdas with the standard query operators

LINQ to Objects, among other implementations, have an input parameter whose type is one of the `Func<TResult>` family of generic delegates. These delegates use type parameters to define the number and type of input parameters, and the return type of the delegate. `Func` delegates are useful for encapsulating user-defined expressions that are applied to each element in a set of source data. For example, consider the `Func<T,TResult>` delegate type:

```
public delegate TResult Func<in T, out TResult>(T arg)
```

The delegate can be instantiated as a `Func<int, bool>` instance where `int` is an input parameter and `bool` is the return value. The return value is always specified in the last type parameter. For example, `Func<int, string, bool>` defines a delegate with two input parameters, `int` and `string`, and a return type of `bool`. The following `Func` delegate, when it's invoked, returns Boolean value that indicates whether the input parameter is equal to five:

```
Func<int, bool> equalsFive = x => x == 5;
bool result = equalsFive(4);
Console.WriteLine(result);    // False
```

You can also supply a lambda expression when the argument type is an [Expression<TDelegate>](#), for example in the standard query operators that are defined in the [Queryable](#) type. When you specify an [Expression<TDelegate>](#) argument, the lambda is compiled to an expression tree.

The following example uses the [Count](#) standard query operator:

```
int[] numbers = { 5, 4, 1, 3, 9, 8, 6, 7, 2, 0 };
int oddNumbers = numbers.Count(n => n % 2 == 1);
Console.WriteLine($"There are {oddNumbers} odd numbers in {string.Join(" ", numbers)}");
```

The compiler can infer the type of the input parameter, or you can also specify it explicitly. This particular lambda expression counts those integers (`n`) which when divided by two have a remainder of 1.

The following example produces a sequence that contains all elements in the `numbers` array that precede the 9, because that's the first number in the sequence that doesn't meet the condition:

```
int[] numbers = { 5, 4, 1, 3, 9, 8, 6, 7, 2, 0 };
var firstNumbersLessThanSix = numbers.TakeWhile(n => n < 6);
Console.WriteLine(string.Join(" ", firstNumbersLessThanSix));
// Output:
// 5 4 1 3
```

The following example specifies multiple input parameters by enclosing them in parentheses. The method returns all the elements in the `numbers` array until it finds a number whose value is less than its ordinal position in the array:

```
int[] numbers = { 5, 4, 1, 3, 9, 8, 6, 7, 2, 0 };
var firstSmallNumbers = numbers.TakeWhile((n, index) => n >= index);
Console.WriteLine(string.Join(" ", firstSmallNumbers));
// Output:
// 5 4
```

You don't use lambda expressions directly in [query expressions](#), but you can use them in method calls within query expressions, as the following example shows:

```

var numberSets = new List<int[]>
{
    new[] { 1, 2, 3, 4, 5 },
    new[] { 0, 0, 0 },
    new[] { 9, 8 },
    new[] { 1, 0, 1, 0, 1, 0, 1, 0 }
};

var setsWithManyPositives =
    from numberSet in numberSets
    where numberSet.Count(n => n > 0) > 3
    select numberSet;

foreach (var numberSet in setsWithManyPositives)
{
    Console.WriteLine(string.Join(" ", numberSet));
}
// Output:
// 1 2 3 4 5
// 1 0 1 0 1 0 1 0

```

Type inference in lambda expressions

When writing lambdas, you often don't have to specify a type for the input parameters because the compiler can infer the type based on the lambda body, the parameter types, and other factors as described in the C# language specification. For most of the standard query operators, the first input is the type of the elements in the source sequence. If you're querying an `IEnumerable<Customer>`, then the input variable is inferred to be a `Customer` object, which means you have access to its methods and properties:

```
customers.Where(c => c.City == "London");
```

The general rules for type inference for lambdas are as follows:

- The lambda must contain the same number of parameters as the delegate type.
- Each input parameter in the lambda must be implicitly convertible to its corresponding delegate parameter.
- The return value of the lambda (if any) must be implicitly convertible to the delegate's return type.

Natural type of a lambda expression

A lambda expression in itself doesn't have a type because the common type system has no intrinsic concept of "lambda expression." However, it's sometimes convenient to speak informally of the "type" of a lambda expression. That informal "type" refers to the delegate type or [Expression](#) type to which the lambda expression is converted.

Beginning with C# 10, a lambda expression may have a *natural type*. Instead of forcing you to declare a delegate type, such as `Func<...>` or `Action<...>` for a lambda expression, the compiler may infer the delegate type from the lambda expression. For example, consider the following declaration:

```
var parse = (string s) => int.Parse(s);
```

The compiler can infer `parse` to be a `Func<string, int>`. The compiler chooses an available `Func` or `Action` delegate, if a suitable one exists. Otherwise, it synthesizes a delegate type. For example, the delegate type is synthesized if the lambda expression has `ref` parameters. When a lambda expression has a natural type, it can be assigned to a less explicit type, such as `System.Object` or `System.Delegate`:

```
object parse = (string s) => int.Parse(s); // Func<string, int>
Delegate parse = (string s) => int.Parse(s); // Func<string, int>
```

Method groups (that is, method names without parameter lists) with exactly one overload have a natural type:

```
var read = Console.Read; // Just one overload; Func<int> inferred
var write = Console.Write; // ERROR: Multiple overloads, can't choose
```

If you assign a lambda expression to [System.Linq.Expressions.LambdaExpression](#), or [System.Linq.Expressions.Expression](#), and the lambda has a natural delegate type, the expression has a natural type of [System.Linq.Expressions.Expression<TDelegate>](#), with the natural delegate type used as the argument for the type parameter:

```
LambdaExpression parseExpr = (string s) => int.Parse(s); // Expression<Func<string, int>>
Expression parseExpr = (string s) => int.Parse(s); // Expression<Func<string, int>>
```

Not all lambda expressions have a natural type. Consider the following declaration:

```
var parse = s => int.Parse(s); // ERROR: Not enough type info in the lambda
```

The compiler can't infer a parameter type for `s`. When the compiler can't infer a natural type, you must declare the type:

```
Func<string, int> parse = s => int.Parse(s);
```

Explicit return type

Typically, the return type of a lambda expression is obvious and inferred. For some expressions, that doesn't work:

```
var choose = (bool b) => b ? 1 : "two"; // ERROR: Can't infer return type
```

Beginning with C# 10, you can specify the return type of a lambda expression before the input parameters. When you specify an explicit return type, you must parenthesize the input parameters:

```
var choose = object (bool b) => b ? 1 : "two"; // Func<bool, object>
```

Attributes

Beginning with C# 10, you can add attributes to a lambda expression and its parameters. The following example shows how to add attributes to a lambda expression:

```
Func<string, int> parse = [Example(1)] (s) => int.Parse(s);
var choose = [Example(2)][Example(3)] object (bool b) => b ? 1 : "two";
```

You can also add attributes to the input parameters or return value, as the following example shows:

```
var sum = ([Example(1)] int a, [Example(2), Example(3)] int b) => a + b;  
var inc = [return: Example(1)] (int s) => s++;
```

As the preceding examples show, you must parenthesize the input parameters when you add attributes to a lambda expression or its parameters.

IMPORTANT

Lambda expressions are invoked through the underlying delegate type. That is different than methods and local functions. The delegate's `Invoke` method doesn't check attributes on the lambda expression. Attributes don't have any effect when the lambda expression is invoked. Attributes on lambda expressions are useful for code analysis, and can be discovered via reflection. One consequence of this decision is that the `System.Diagnostics.ConditionalAttribute` cannot be applied to a lambda expression.

Capture of outer variables and variable scope in lambda expressions

Lambdas can refer to *outer variables*. These are the variables that are in scope in the method that defines the lambda expression, or in scope in the type that contains the lambda expression. Variables that are captured in this manner are stored for use in the lambda expression even if the variables would otherwise go out of scope and be garbage collected. An outer variable must be definitely assigned before it can be consumed in a lambda expression. The following example demonstrates these rules:

```

public static class VariableScopeWithLambdas
{
    public class VariableCaptureGame
    {
        internal Action<int> updateCapturedLocalVariable;
        internal Func<int, bool> isEqualToCapturedLocalVariable;

        public void Run(int input)
        {
            int j = 0;

            updateCapturedLocalVariable = x =>
            {
                j = x;
                bool result = j > input;
                Console.WriteLine($"{j} is greater than {input}: {result}");
            };

            isEqualToCapturedLocalVariable = x => x == j;

            Console.WriteLine($"Local variable before lambda invocation: {j}");
            updateCapturedLocalVariable(10);
            Console.WriteLine($"Local variable after lambda invocation: {j}");
        }
    }

    public static void Main()
    {
        var game = new VariableCaptureGame();

        int gameInput = 5;
        game.Run(gameInput);

        int jTry = 10;
        bool result = game.isEqualToCapturedLocalVariable(jTry);
        Console.WriteLine($"Captured local variable is equal to {jTry}: {result}");

        int anotherJ = 3;
        game.updateCapturedLocalVariable(anotherJ);

        bool equalToAnother = game.isEqualToCapturedLocalVariable(anotherJ);
        Console.WriteLine($"Another lambda observes a new value of captured variable: {equalToAnother}");
    }
    // Output:
    // Local variable before lambda invocation: 0
    // 10 is greater than 5: True
    // Local variable after lambda invocation: 10
    // Captured local variable is equal to 10: True
    // 3 is greater than 5: False
    // Another lambda observes a new value of captured variable: True
}

```

The following rules apply to variable scope in lambda expressions:

- A variable that is captured won't be garbage-collected until the delegate that references it becomes eligible for garbage collection.
- Variables introduced within a lambda expression aren't visible in the enclosing method.
- A lambda expression can't directly capture an [in](#), [ref](#), or [out](#) parameter from the enclosing method.
- A [return](#) statement in a lambda expression doesn't cause the enclosing method to return.
- A lambda expression can't contain a [goto](#), [break](#), or [continue](#) statement if the target of that jump statement is outside the lambda expression block. It's also an error to have a jump statement outside the lambda expression block if the target is inside the block.

Beginning with C# 9.0, you can apply the `static` modifier to a lambda expression to prevent unintentional capture of local variables or instance state by the lambda:

```
Func<double, double> square = static x => x * x;
```

A static lambda can't capture local variables or instance state from enclosing scopes, but may reference static members and constant definitions.

C# language specification

For more information, see the [Anonymous function expressions](#) section of the [C# language specification](#).

For more information about features added in C# 9.0 and later, see the following feature proposal notes:

- [Lambda discard parameters \(C# 9.0\)](#)
- [Static anonymous functions \(C# 9.0\)](#)
- [Lambda improvements \(C# 10\)](#)

See also

- [C# reference](#)
- [C# operators and expressions](#)
- [LINQ \(Language-Integrated Query\)](#)
- [Expression trees](#)
- [Local functions vs. lambda expressions](#)
- [LINQ sample queries](#)
- [XQuery sample](#)
- [101 LINQ samples](#)

Patterns (C# reference)

12/28/2021 • 14 minutes to read • [Edit Online](#)

C# introduced pattern matching in C# 7.0. Since then, each major C# version extends pattern matching capabilities. The following C# expressions and statements support pattern matching:

- `is` [expression](#)
- `switch` [statement](#)
- `switch` [expression](#) (introduced in C# 8.0)

In those constructs, you can match an input expression against any of the following patterns:

- [Declaration pattern](#): to check the run-time type of an expression and, if a match succeeds, assign an expression result to a declared variable. Introduced in C# 7.0.
- [Type pattern](#): to check the run-time type of an expression. Introduced in C# 9.0.
- [Constant pattern](#): to test if an expression result equals a specified constant. Introduced in C# 7.0.
- [Relational patterns](#): to compare an expression result with a specified constant. Introduced in C# 9.0.
- [Logical patterns](#): to test if an expression matches a logical combination of patterns. Introduced in C# 9.0.
- [Property pattern](#): to test if an expression's properties or fields match nested patterns. Introduced in C# 8.0.
- [Positional pattern](#): to deconstruct an expression result and test if the resulting values match nested patterns. Introduced in C# 8.0.
- `var` [pattern](#): to match any expression and assign its result to a declared variable. Introduced in C# 7.0.
- [Discard pattern](#): to match any expression. Introduced in C# 8.0.

[Logical](#), [property](#), and [positional](#) patterns are *recursive* patterns. That is, they can contain *nested* patterns.

For the example of how to use those patterns to build a data-driven algorithm, see [Tutorial: Use pattern matching to build type-driven and data-driven algorithms](#).

Declaration and type patterns

You use declaration and type patterns to check if the run-time type of an expression is compatible with a given type. With a declaration pattern, you can also declare a new local variable. When a declaration pattern matches an expression, that variable is assigned a converted expression result, as the following example shows:

```
object greeting = "Hello, World!";
if (greeting is string message)
{
    Console.WriteLine(message.ToLower()); // output: hello, world!
}
```

Beginning with C# 7.0, a *declaration pattern* with type `T` matches an expression when an expression result is non-null and any of the following conditions are true:

- The run-time type of an expression result is `T`.
- The run-time type of an expression result derives from type `T`, implements interface `T`, or another [implicit reference conversion](#) exists from it to `T`. The following example demonstrates two cases when this condition is true:

```

var numbers = new int[] { 10, 20, 30 };
Console.WriteLine(GetSourceLabel(numbers)); // output: 1

var letters = new List<char> { 'a', 'b', 'c', 'd' };
Console.WriteLine(GetSourceLabel(letters)); // output: 2

static int GetSourceLabel<T>(IEnumerable<T> source) => source switch
{
    Array array => 1,
    ICollection<T> collection => 2,
    _ => 3,
};

```

In the preceding example, at the first call to the `GetSourceLabel` method, the first pattern matches an argument value because the argument's run-time type `int[]` derives from the `Array` type. At the second call to the `GetSourceLabel` method, the argument's run-time type `List<T>` doesn't derive from the `Array` type but implements the `ICollection<T>` interface.

- The run-time type of an expression result is a [nullable value type](#) with the underlying type `T`.
- A [boxing](#) or [unboxing](#) conversion exists from the run-time type of an expression result to type `T`.

The following example demonstrates the last two conditions:

```

int? xNullable = 7;
int y = 23;
object yBoxed = y;
if (xNullable is int a && yBoxed is int b)
{
    Console.WriteLine(a + b); // output: 30
}

```

If you want to check only the type of an expression, you can use a discard `_` in place of a variable's name, as the following example shows:

```

public abstract class Vehicle {}
public class Car : Vehicle {}
public class Truck : Vehicle {}

public static class TollCalculator
{
    public static decimal CalculateToll(this Vehicle vehicle) => vehicle switch
    {
        Car _ => 2.00m,
        Truck _ => 7.50m,
        null => throw new ArgumentNullException(nameof(vehicle)),
        _ => throw new ArgumentException("Unknown type of a vehicle", nameof(vehicle)),
    };
}

```

Beginning with C# 9.0, for that purpose you can use a *type pattern*, as the following example shows:

```

public static decimal CalculateToll(this Vehicle vehicle) => vehicle switch
{
    Car => 2.00m,
    Truck => 7.50m,
    null => throw new ArgumentNullException(nameof(vehicle)),
    _ => throw new ArgumentException("Unknown type of a vehicle", nameof(vehicle)),
};

```

Like a declaration pattern, a type pattern matches an expression when an expression result is non-null and its run-time type satisfies any of the conditions listed above.

For more information, see the [Declaration pattern](#) and [Type pattern](#) sections of the feature proposal notes.

Constant pattern

Beginning with C# 7.0, you use a *constant pattern* to test if an expression result equals a specified constant, as the following example shows:

```
public static decimal GetGroupTicketPrice(int visitorCount) => visitorCount switch
{
    1 => 12.0m,
    2 => 20.0m,
    3 => 27.0m,
    4 => 32.0m,
    0 => 0.0m,
    _ => throw new ArgumentException($"Not supported number of visitors: {visitorCount}",
    nameof(visitorCount)),
};
```

In a constant pattern, you can use any constant expression, such as:

- an [integer](#) or [floating-point](#) numerical literal
- a [char](#) or a [string](#) literal
- a Boolean value `true` or `false`
- an [enum](#) value
- the name of a declared [const](#) field or local
- `null`

Use a constant pattern to check for `null`, as the following example shows:

```
if (input is null)
{
    return;
}
```

The compiler guarantees that no user-overloaded equality operator `==` is invoked when expression `x is null` is evaluated.

Beginning with C# 9.0, you can use a [negated](#) `null` constant pattern to check for non-null, as the following example shows:

```
if (input is not null)
{
    // ...
}
```

For more information, see the [Constant pattern](#) section of the feature proposal note.

Relational patterns

Beginning with C# 9.0, you use a *relational pattern* to compare an expression result with a constant, as the following example shows:

```

Console.WriteLine(Classify(13)); // output: Too high
Console.WriteLine(Classify(double.NaN)); // output: Unknown
Console.WriteLine(Classify(2.4)); // output: Acceptable

static string Classify(double measurement) => measurement switch
{
    < -4.0 => "Too low",
    > 10.0 => "Too high",
    double.NaN => "Unknown",
    _ => "Acceptable",
};

```

In a relational pattern, you can use any of the [relational operators](#) `<`, `>`, `<=`, or `>=`. The right-hand part of a relational pattern must be a constant expression. The constant expression can be of an [integer](#), [floating-point](#), [char](#), or [enum](#) type.

To check if an expression result is in a certain range, match it against a [conjunctive](#) `and` pattern, as the following example shows:

```

Console.WriteLine(GetCalendarSeason(new DateTime(2021, 3, 14))); // output: spring
Console.WriteLine(GetCalendarSeason(new DateTime(2021, 7, 19))); // output: summer
Console.WriteLine(GetCalendarSeason(new DateTime(2021, 2, 17))); // output: winter

static string GetCalendarSeason(DateTime date) => date.Month switch
{
    >= 3 and < 6 => "spring",
    >= 6 and < 9 => "summer",
    >= 9 and < 12 => "autumn",
    12 or (>= 1 and < 3) => "winter",
    _ => throw new ArgumentOutOfRangeException(nameof(date), $"Date with unexpected month: {date.Month}."),
};

```

If an expression result is `null` or fails to convert to the type of a constant by a nullable or unboxing conversion, a relational pattern doesn't match an expression.

For more information, see the [Relational patterns](#) section of the feature proposal note.

Logical patterns

Beginning with C# 9.0, you use the `not`, `and`, and `or` pattern combinators to create the following *logical patterns*:

- *Negation* `not` pattern that matches an expression when the negated pattern doesn't match the expression. The following example shows how you can negate a [constant](#) `null` pattern to check if an expression is non-null:

```

if (input is not null)
{
    // ...
}

```

- *Conjunctive* `and` pattern that matches an expression when both patterns match the expression. The following example shows how you can combine [relational patterns](#) to check if a value is in a certain range:

```

Console.WriteLine(Classify(13)); // output: High
Console.WriteLine(Classify(-100)); // output: Too low
Console.WriteLine(Classify(5.7)); // output: Acceptable

static string Classify(double measurement) => measurement switch
{
    < -40.0 => "Too low",
    >= -40.0 and < 0 => "Low",
    >= 0 and < 10.0 => "Acceptable",
    >= 10.0 and < 20.0 => "High",
    >= 20.0 => "Too high",
    double.NaN => "Unknown",
};

```

- *Disjunctive* `or` pattern that matches an expression when either of patterns matches the expression, as the following example shows:

```

Console.WriteLine(GetCalendarSeason(new DateTime(2021, 1, 19))); // output: winter
Console.WriteLine(GetCalendarSeason(new DateTime(2021, 10, 9))); // output: autumn
Console.WriteLine(GetCalendarSeason(new DateTime(2021, 5, 11))); // output: spring

static string GetCalendarSeason(DateTime date) => date.Month switch
{
    3 or 4 or 5 => "spring",
    6 or 7 or 8 => "summer",
    9 or 10 or 11 => "autumn",
    12 or 1 or 2 => "winter",
    _ => throw new ArgumentOutOfRangeException(nameof(date), $"Date with unexpected month: {date.Month}."),
};

```

As the preceding example shows, you can repeatedly use the pattern combinators in a pattern.

Precedence and order of checking

The following list orders pattern combinators starting from the highest precedence to the lowest:

- `not`
- `and`
- `or`

To explicitly specify the precedence, use parentheses, as the following example shows:

```

static bool IsLetter(char c) => c is (>= 'a' and <= 'z') or (>= 'A' and <= 'Z');

```

NOTE

The order in which patterns are checked is undefined. At run time, the right-hand nested patterns of `or` and `and` patterns can be checked first.

For more information, see the [Pattern combinators](#) section of the feature proposal note.

Property pattern

Beginning with C# 8.0, you use a *property pattern* to match an expression's properties or fields against nested patterns, as the following example shows:

```
static bool IsConferenceDay(DateTime date) => date is { Year: 2020, Month: 5, Day: 19 or 20 or 21 };
```

A property pattern matches an expression when an expression result is non-null and every nested pattern matches the corresponding property or field of the expression result.

You can also add a run-time type check and a variable declaration to a property pattern, as the following example shows:

```
Console.WriteLine(TakeFive("Hello, world!")); // output: Hello
Console.WriteLine(TakeFive("Hi!")); // output: Hi!
Console.WriteLine(TakeFive(new[] { '1', '2', '3', '4', '5', '6', '7' })); // output: 12345
Console.WriteLine(TakeFive(new[] { 'a', 'b', 'c' })); // output: abc

static string TakeFive(object input) => input switch
{
    string { Length: >= 5 } s => s.Substring(0, 5),
    string s => s,

    ICollection<char> { Count: >= 5 } symbols => new string(symbols.Take(5).ToArray()),
    ICollection<char> symbols => new string(symbols.ToArray()),

    null => throw new ArgumentNullException(nameof(input)),
    _ => throw new ArgumentException("Not supported input type."),
};
```

A property pattern is a recursive pattern. That is, you can use any pattern as a nested pattern. Use a property pattern to match parts of data against nested patterns, as the following example shows:

```
public record Point(int X, int Y);
public record Segment(Point Start, Point End);

static bool IsAnyEndOnXAxis(Segment segment) =>
    segment is { Start: { Y: 0 } } or { End: { Y: 0 } };
```

The preceding example uses two features available in C# 9.0 and later: `or` [pattern combinator](#) and [record types](#).

Beginning with C# 10, you can reference nested properties or fields within a property pattern. For example, you can refactor the method from the preceding example into the following equivalent code:

```
static bool IsAnyEndOnXAxis(Segment segment) =>
    segment is { Start.Y: 0 } or { End.Y: 0 };
```

For more information, see the [Property pattern](#) section of the feature proposal note and the [Extended property patterns](#) feature proposal note.

Positional pattern

Beginning with C# 8.0, you use a *positional pattern* to deconstruct an expression result and match the resulting values against the corresponding nested patterns, as the following example shows:

```

public readonly struct Point
{
    public int X { get; }
    public int Y { get; }

    public Point(int x, int y) => (X, Y) = (x, y);

    public void Deconstruct(out int x, out int y) => (x, y) = (X, Y);
}

static string Classify(Point point) => point switch
{
    (0, 0) => "Origin",
    (1, 0) => "positive X basis end",
    (0, 1) => "positive Y basis end",
    _ => "Just a point",
};

```

At the preceding example, the type of an expression contains the [Deconstruct](#) method, which is used to deconstruct an expression result. You can also match expressions of [tuple types](#) against positional patterns. In that way, you can match multiple inputs against various patterns, as the following example shows:

```

static decimal GetGroupTicketPriceDiscount(int groupSize, DateTime visitDate)
=> (groupSize, visitDate.DayOfWeek) switch
{
    (<= 0, _) => throw new ArgumentException("Group size must be positive."),
    (_, DayOfWeek.Saturday or DayOfWeek.Sunday) => 0.0m,
    (>= 5 and < 10, DayOfWeek.Monday) => 20.0m,
    (>= 10, DayOfWeek.Monday) => 30.0m,
    (>= 5 and < 10, _) => 12.0m,
    (>= 10, _) => 15.0m,
    _ => 0.0m,
};

```

The preceding example uses [relational](#) and [logical](#) patterns, which are available in C# 9.0 and later.

You can use the names of tuple elements and `Deconstruct` parameters in a positional pattern, as the following example shows:

```

var numbers = new List<int> { 1, 2, 3 };
if (SumAndCount(numbers) is (Sum: var sum, Count: > 0))
{
    Console.WriteLine($"Sum of [{string.Join(" ", numbers)}] is {sum}"); // output: Sum of [1 2 3] is 6
}

static (double Sum, int Count) SumAndCount(IEnumerable<int> numbers)
{
    int sum = 0;
    int count = 0;
    foreach (int number in numbers)
    {
        sum += number;
        count++;
    }
    return (sum, count);
}

```

You can also extend a positional pattern in any of the following ways:

- Add a run-time type check and a variable declaration, as the following example shows:


```

public record Point2D(int X, int Y);
public record Point3D(int X, int Y, int Z);

static string PrintIfAllCoordinatesArePositive(object point) => point switch
{
    Point2D (> 0, > 0) p => p.ToString(),
    Point3D (> 0, > 0, > 0) p => p.ToString(),
    _ => string.Empty,
};

```

The preceding example uses [positional records](#) that implicitly provide the `Deconstruct` method.

- Use a [property pattern](#) within a positional pattern, as the following example shows:

```

public record WeightedPoint(int X, int Y)
{
    public double Weight { get; set; }
}

static bool IsInDomain(WeightedPoint point) => point is (>= 0, >= 0) { Weight: >= 0.0 };

```

- Combine two preceding usages, as the following example shows:

```

if (input is WeightedPoint (> 0, > 0) { Weight: > 0.0 } p)
{
    // ..
}

```

A positional pattern is a recursive pattern. That is, you can use any pattern as a nested pattern.

For more information, see the [Positional pattern](#) section of the feature proposal note.

var pattern

Beginning with C# 7.0, you use a `var` *pattern* to match any expression, including `null`, and assign its result to a new local variable, as the following example shows:

```

static bool IsAcceptable(int id, int absLimit) =>
    SimulateDataFetch(id) is var results
    && results.Min() >= -absLimit
    && results.Max() <= absLimit;

static int[] SimulateDataFetch(int id)
{
    var rand = new Random();
    return Enumerable
        .Range(start: 0, count: 5)
        .Select(s => rand.Next(minValue: -10, maxValue: 11))
        .ToArray();
}

```

A `var` pattern is useful when you need a temporary variable within a Boolean expression to hold the result of intermediate calculations. You can also use a `var` pattern when you need to perform additional checks in `when` case guards of a `switch` expression or statement, as the following example shows:

```
public record Point(int X, int Y);

static Point Transform(Point point) => point switch
{
    var (x, y) when x < y => new Point(-x, y),
    var (x, y) when x > y => new Point(x, -y),
    var (x, y) => new Point(x, y),
};

static void TestTransform()
{
    Console.WriteLine(Transform(new Point(1, 2))); // output: Point { X = -1, Y = 2 }
    Console.WriteLine(Transform(new Point(5, 2))); // output: Point { X = 5, Y = -2 }
}
```

In the preceding example, pattern `var (x, y)` is equivalent to a [positional pattern](#) `(var x, var y)`.

In a `var` pattern, the type of a declared variable is the compile-time type of the expression that is matched against the pattern.

For more information, see the [Var pattern](#) section of the feature proposal note.

Discard pattern

Beginning with C# 8.0, you use a *discard pattern* `_` to match any expression, including `null`, as the following example shows:

```
Console.WriteLine(GetDiscountInPercent(DayOfWeek.Friday)); // output: 5.0
Console.WriteLine(GetDiscountInPercent(null)); // output: 0.0
Console.WriteLine(GetDiscountInPercent((DayOfWeek)10)); // output: 0.0

static decimal GetDiscountInPercent(DayOfWeek? dayOfWeek) => dayOfWeek switch
{
    DayOfWeek.Monday => 0.5m,
    DayOfWeek.Tuesday => 12.5m,
    DayOfWeek.Wednesday => 7.5m,
    DayOfWeek.Thursday => 12.5m,
    DayOfWeek.Friday => 5.0m,
    DayOfWeek.Saturday => 2.5m,
    DayOfWeek.Sunday => 2.0m,
    _ => 0.0m,
};
```

In the preceding example, a discard pattern is used to handle `null` and any integer value that doesn't have the corresponding member of the [DayOfWeek](#) enumeration. That guarantees that a `switch` expression in the example handles all possible input values. If you don't use a discard pattern in a `switch` expression and none of the expression's patterns matches an input, the runtime [throws an exception](#). The compiler generates a warning if a `switch` expression doesn't handle all possible input values.

A discard pattern cannot be a pattern in an `is` expression or a `switch` statement. In those cases, to match any expression, use a [var pattern](#) with a discard: `var _`.

For more information, see the [Discard pattern](#) section of the feature proposal note.

Parenthesized pattern

Beginning with C# 9.0, you can put parentheses around any pattern. Typically, you do that to emphasize or change the precedence in [logical patterns](#), as the following example shows:

```
if (input is not (float or double))
{
    return;
}
```

C# language specification

For more information, see the following feature proposal notes:

- [Pattern matching for C# 7.0](#)
- [Recursive pattern matching \(introduced in C# 8.0\)](#)
- [Pattern-matching changes for C# 9.0](#)
- [Extended property patterns \(C# 10\)](#)

See also

- [C# reference](#)
- [C# operators and expressions](#)
- [Tutorial: Use pattern matching to build type-driven and data-driven algorithms](#)

+ and += operators (C# reference)

12/28/2021 • 2 minutes to read • [Edit Online](#)

The `+` and `+=` operators are supported by the built-in [integral](#) and [floating-point](#) numeric types, the [string](#) type, and [delegate](#) types.

For information about the arithmetic `+` operator, see the [Unary plus and minus operators](#) and [Addition operator](#) + sections of the [Arithmetic operators](#) article.

String concatenation

When one or both operands are of type [string](#), the `+` operator concatenates the string representations of its operands (the string representation of `null` is an empty string):

```
Console.WriteLine("Forgot" + "white space");
Console.WriteLine("Probably the oldest constant: " + Math.PI);
Console.WriteLine(null + "Nothing to add.");
// Output:
// Forgotwhite space
// Probably the oldest constant: 3.14159265358979
// Nothing to add.
```

Beginning with C# 6, [string interpolation](#) provides a more convenient way to format strings:

```
Console.WriteLine($"Probably the oldest constant: {Math.PI:F2}");
// Output:
// Probably the oldest constant: 3.14
```

Beginning with C# 10, you can use string interpolation to initialize a constant string when all the expressions used for placeholders are also constant strings.

Delegate combination

For operands of the same [delegate](#) type, the `+` operator returns a new delegate instance that, when invoked, invokes the left-hand operand and then invokes the right-hand operand. If any of the operands is `null`, the `+` operator returns the value of another operand (which also might be `null`). The following example shows how delegates can be combined with the `+` operator:

```
Action a = () => Console.Write("a");
Action b = () => Console.Write("b");
Action ab = a + b;
ab(); // output: ab
```

To perform delegate removal, use the `-` operator.

For more information about delegate types, see [Delegates](#).

Addition assignment operator +=

An expression using the `+=` operator, such as

```
x += y
```

is equivalent to

```
x = x + y
```

except that `x` is only evaluated once.

The following example demonstrates the usage of the `+=` operator:

```
int i = 5;
i += 9;
Console.WriteLine(i);
// Output: 14

string story = "Start. ";
story += "End.";
Console.WriteLine(story);
// Output: Start. End.

Action printer = () => Console.Write("a");
printer(); // output: a

Console.WriteLine();
printer += () => Console.Write("b");
printer(); // output: ab
```

You also use the `+=` operator to specify an event handler method when you subscribe to an [event](#). For more information, see [How to: subscribe to and unsubscribe from events](#).

Operator overloadability

A user-defined type can [overload](#) the `+` operator. When a binary `+` operator is overloaded, the `+=` operator is also implicitly overloaded. A user-defined type cannot explicitly overload the `+=` operator.

C# language specification

For more information, see the [Unary plus operator](#) and [Addition operator](#) sections of the [C# language specification](#).

See also

- [C# reference](#)
- [C# operators and expressions](#)
- [How to concatenate multiple strings](#)
- [Events](#)
- [Arithmetic operators](#)
- [- and -= operators](#)

- and -= operators (C# reference)

12/28/2021 • 3 minutes to read • [Edit Online](#)

The `-` and `-=` operators are supported by the built-in [integral](#) and [floating-point](#) numeric types and [delegate](#) types.

For information about the arithmetic `-` operator, see the [Unary plus and minus operators](#) and [Subtraction operator](#) - sections of the [Arithmetic operators](#) article.

Delegate removal

For operands of the same [delegate](#) type, the `-` operator returns a delegate instance that is calculated as follows:

- If both operands are non-null and the invocation list of the right-hand operand is a proper contiguous sublist of the invocation list of the left-hand operand, the result of the operation is a new invocation list that is obtained by removing the right-hand operand's entries from the invocation list of the left-hand operand. If the right-hand operand's list matches multiple contiguous sublists in the left-hand operand's list, only the right-most matching sublist is removed. If removal results in an empty list, the result is `null`.

```
Action a = () => Console.Write("a");
Action b = () => Console.Write("b");

var abbaab = a + b + b + a + a + b;
abbaab(); // output: abbaab
Console.WriteLine();

var ab = a + b;
var abba = abbaab - ab;
abba(); // output: abba
Console.WriteLine();

var nihil = abbaab - abbaab;
Console.WriteLine(nihil is null); // output: True
```

- If the invocation list of the right-hand operand is not a proper contiguous sublist of the invocation list of the left-hand operand, the result of the operation is the left-hand operand. For example, removing a delegate that is not part of the multicast delegate does nothing and results in the unchanged multicast delegate.

```

Action a = () => Console.Write("a");
Action b = () => Console.Write("b");

var abbaab = a + b + b + a + a + b;
var aba = a + b + a;

var first = abbaab - aba;
first(); // output: abbaab
Console.WriteLine();
Console.WriteLine(object.ReferenceEquals(abbaab, first)); // output: True

Action a2 = () => Console.Write("a");
var changed = aba - a;
changed(); // output: ab
Console.WriteLine();
var unchanged = aba - a2;
unchanged(); // output: aba
Console.WriteLine();
Console.WriteLine(object.ReferenceEquals(aba, unchanged)); // output: True

```

The preceding example also demonstrates that during delegate removal delegate instances are compared. For example, delegates that are produced from evaluation of identical [lambda expressions](#) are not equal. For more information about delegate equality, see the [Delegate equality operators](#) section of the [C# language specification](#).

- If the left-hand operand is `null`, the result of the operation is `null`. If the right-hand operand is `null`, the result of the operation is the left-hand operand.

```

Action a = () => Console.Write("a");

var nothing = null - a;
Console.WriteLine(nothing is null); // output: True

var first = a - null;
a(); // output: a
Console.WriteLine();
Console.WriteLine(object.ReferenceEquals(first, a)); // output: True

```

To combine delegates, use the [+ operator](#).

For more information about delegate types, see [Delegates](#).

Subtraction assignment operator -=

An expression using the `-=` operator, such as

```
x -= y
```

is equivalent to

```
x = x - y
```

except that `x` is only evaluated once.

The following example demonstrates the usage of the `-=` operator:

```
int i = 5;
i -= 9;
Console.WriteLine(i);
// Output: -4

Action a = () => Console.Write("a");
Action b = () => Console.Write("b");
var printer = a + b + a;
printer(); // output: aba

Console.WriteLine();
printer -= a;
printer(); // output: ab
```

You also use the `-=` operator to specify an event handler method to remove when you unsubscribe from an [event](#). For more information, see [How to subscribe to and unsubscribe from events](#).

Operator overloadability

A user-defined type can [overload](#) the `-` operator. When a binary `-` operator is overloaded, the `-=` operator is also implicitly overloaded. A user-defined type cannot explicitly overload the `-=` operator.

C# language specification

For more information, see the [Unary minus operator](#) and [Subtraction operator](#) sections of the [C# language specification](#).

See also

- [C# reference](#)
- [C# operators and expressions](#)
- [Events](#)
- [Arithmetic operators](#)
- [+ and += operators](#)

?: operator (C# reference)

12/28/2021 • 3 minutes to read • [Edit Online](#)

The conditional operator `?:`, also known as the ternary conditional operator, evaluates a Boolean expression and returns the result of one of the two expressions, depending on whether the Boolean expression evaluates to `true` or `false`, as the following example shows:

```
string GetWeatherDisplay(double tempInCelsius) => tempInCelsius < 20.0 ? "Cold." : "Perfect!";

Console.WriteLine(GetWeatherDisplay(15)); // output: Cold.
Console.WriteLine(GetWeatherDisplay(27)); // output: Perfect!
```

As the preceding example shows, the syntax for the conditional operator is as follows:

```
condition ? consequent : alternative
```

The `condition` expression must evaluate to `true` or `false`. If `condition` evaluates to `true`, the `consequent` expression is evaluated, and its result becomes the result of the operation. If `condition` evaluates to `false`, the `alternative` expression is evaluated, and its result becomes the result of the operation. Only `consequent` or `alternative` is evaluated.

Beginning with C# 9.0, conditional expressions are target-typed. That is, if a target type of a conditional expression is known, the types of `consequent` and `alternative` must be implicitly convertible to the target type, as the following example shows:

```
var rand = new Random();
var condition = rand.NextDouble() > 0.5;

int? x = condition ? 12 : null;

IEnumerable<int> xs = x is null ? new List<int>() { 0, 1 } : new int[] { 2, 3 };
```

If a target type of a conditional expression is unknown (for example, when you use the `var` keyword) or in C# 8.0 and earlier, the type of `consequent` and `alternative` must be the same or there must be an implicit conversion from one type to the other:

```
var rand = new Random();
var condition = rand.NextDouble() > 0.5;

var x = condition ? 12 : (int?)null;
```

The conditional operator is right-associative, that is, an expression of the form

```
a ? b : c ? d : e
```

is evaluated as

```
a ? b : (c ? d : e)
```

TIP

You can use the following mnemonic device to remember how the conditional operator is evaluated:

```
is this condition true ? yes : no
```

Conditional ref expression

Beginning with C# 7.2, a [ref local](#) or [ref readonly local](#) variable can be assigned conditionally with a conditional ref expression. You can also use a conditional ref expression as a [reference return value](#) or as a [ref method argument](#).

The syntax for a conditional ref expression is as follows:

```
condition ? ref consequent : ref alternative
```

Like the original conditional operator, a conditional ref expression evaluates only one of the two expressions: either `consequent` or `alternative`.

In the case of a conditional ref expression, the type of `consequent` and `alternative` must be the same. Conditional ref expressions are not target-typed.

The following example demonstrates the usage of a conditional ref expression:

```
var smallArray = new int[] { 1, 2, 3, 4, 5 };
var largeArray = new int[] { 10, 20, 30, 40, 50 };

int index = 7;
ref int refValue = ref ((index < 5) ? ref smallArray[index] : ref largeArray[index - 5]);
refValue = 0;

index = 2;
((index < 5) ? ref smallArray[index] : ref largeArray[index - 5]) = 100;

Console.WriteLine(string.Join(" ", smallArray));
Console.WriteLine(string.Join(" ", largeArray));
// Output:
// 1 2 100 4 5
// 10 20 0 40 50
```

Conditional operator and an `if` statement

Use of the conditional operator instead of an `if` statement might result in more concise code in cases when you need conditionally to compute a value. The following example demonstrates two ways to classify an integer as negative or nonnegative:

```
int input = new Random().Next(-5, 5);

string classify;
if (input >= 0)
{
    classify = "nonnegative";
}
else
{
    classify = "negative";
}

classify = (input >= 0) ? "nonnegative" : "negative";
```

Operator overloadability

A user-defined type cannot overload the conditional operator.

C# language specification

For more information, see the [Conditional operator](#) section of the [C# language specification](#).

For more information about features added in C# 7.2 and later, see the following feature proposal notes:

- [Conditional ref expressions \(C# 7.2\)](#)
- [Target-typed conditional expression \(C# 9.0\)](#)

See also

- [C# reference](#)
- [C# operators and expressions](#)
- [if statement](#)
- [?. and ?\[\] operators](#)
- [?? and ??= operators](#)
- [ref keyword](#)

! (null-forgiving) operator (C# reference)

12/28/2021 • 2 minutes to read • [Edit Online](#)

Available in C# 8.0 and later, the unary postfix `!` operator is the null-forgiving, or null-suppression, operator. In an enabled [nullable annotation context](#), you use the null-forgiving operator to declare that expression `x` of a reference type isn't `null`: `x!`. The unary prefix `!` operator is the [logical negation operator](#).

The null-forgiving operator has no effect at run time. It only affects the compiler's static flow analysis by changing the null state of the expression. At run time, expression `x!` evaluates to the result of the underlying expression `x`.

For more information about the nullable reference types feature, see [Nullable reference types](#).

Examples

One of the use cases of the null-forgiving operator is in testing the argument validation logic. For example, consider the following class:

```
#nullable enable
public class Person
{
    public Person(string name) => Name = name ?? throw new ArgumentNullException(nameof(name));

    public string Name { get; }
}
```

Using the [MSTest test framework](#), you can create the following test for the validation logic in the constructor:

```
[TestMethod, ExpectedException(typeof(ArgumentNullException))]
public void NullNameShouldThrowTest()
{
    var person = new Person(null!);
}
```

Without the null-forgiving operator, the compiler generates the following warning for the preceding code:

Warning CS8625: Cannot convert null literal to non-nullable reference type. By using the null-forgiving operator, you inform the compiler that passing `null` is expected and shouldn't be warned about.

You can also use the null-forgiving operator when you definitely know that an expression cannot be `null` but the compiler doesn't manage to recognize that. In the following example, if the `IsValid` method returns `true`, its argument is not `null` and you can safely dereference it:

```
public static void Main()
{
    Person? p = Find("John");
    if (IsValid(p))
    {
        Console.WriteLine($"Found {p!.Name}");
    }
}

public static bool IsValid(Person? person)
    => person is not null && person.Name is not null;
```

Without the null-forgiving operator, the compiler generates the following warning for the `p.Name` code:

```
Warning CS8602: Dereference of a possibly null reference .
```

If you can modify the `IsValid` method, you can use the [NotNullWhen](#) attribute to inform the compiler that an argument of the `IsValid` method cannot be `null` when the method returns `true`:

```
public static void Main()
{
    Person? p = Find("John");
    if (IsValid(p))
    {
        Console.WriteLine($"Found {p.Name}");
    }
}

public static bool IsValid([NotNullWhen(true)] Person? person)
    => person is not null && person.Name is not null;
```

In the preceding example, you don't need to use the null-forgiving operator because the compiler has enough information to find out that `p` cannot be `null` inside the `if` statement. For more information about the attributes that allow you to provide additional information about the null state of a variable, see [Upgrade APIs with attributes to define null expectations](#).

C# language specification

For more information, see [The null-forgiving operator](#) section of the [draft of the nullable reference types specification](#).

See also

- [C# reference](#)
- [C# operators and expressions](#)
- [Tutorial: Design with nullable reference types](#)

?? and ??= operators (C# reference)

12/28/2021 • 2 minutes to read • [Edit Online](#)

The null-coalescing operator `??` returns the value of its left-hand operand if it isn't `null`; otherwise, it evaluates the right-hand operand and returns its result. The `??` operator doesn't evaluate its right-hand operand if the left-hand operand evaluates to non-null.

Available in C# 8.0 and later, the null-coalescing assignment operator `??=` assigns the value of its right-hand operand to its left-hand operand only if the left-hand operand evaluates to `null`. The `??=` operator doesn't evaluate its right-hand operand if the left-hand operand evaluates to non-null.

```
List<int> numbers = null;
int? a = null;

(numbers ??= new List<int>()).Add(5);
Console.WriteLine(string.Join(" ", numbers)); // output: 5

numbers.Add(a ??= 0);
Console.WriteLine(string.Join(" ", numbers)); // output: 5 0
Console.WriteLine(a); // output: 0
```

The left-hand operand of the `??=` operator must be a variable, a [property](#), or an [indexer](#) element.

In C# 7.3 and earlier, the type of the left-hand operand of the `??` operator must be either a [reference type](#) or a [nullable value type](#). Beginning with C# 8.0, that requirement is replaced with the following: the type of the left-hand operand of the `??` and `??=` operators cannot be a non-nullable value type. In particular, beginning with C# 8.0, you can use the null-coalescing operators with unconstrained type parameters:

```
private static void Display<T>(T a, T backup)
{
    Console.WriteLine(a ?? backup);
}
```

The null-coalescing operators are right-associative. That is, expressions of the form

```
a ?? b ?? c
d ??= e ??= f
```

are evaluated as

```
a ?? (b ?? c)
d ??= (e ??= f)
```

Examples

The `??` and `??=` operators can be useful in the following scenarios:

- In expressions with the [null-conditional operators](#) `?.` and `?[]`, you can use the `??` operator to provide an alternative expression to evaluate in case the result of the expression with null-conditional operations is `null`:

```
double SumNumbers(List<double[]> setsOfNumbers, int indexOfSetToSum)
{
    return setsOfNumbers?[indexOfSetToSum]?.Sum() ?? double.NaN;
}

var sum = SumNumbers(null, 0);
Console.WriteLine(sum); // output: NaN
```

- When you work with [nullable value types](#) and need to provide a value of an underlying value type, use the `??` operator to specify the value to provide in case a nullable type value is `null`:

```
int? a = null;
int b = a ?? -1;
Console.WriteLine(b); // output: -1
```

Use the [Nullable<T>.GetValueOrDefault\(\)](#) method if the value to be used when a nullable type value is `null` should be the default value of the underlying value type.

- Beginning with C# 7.0, you can use a [throw expression](#) as the right-hand operand of the `??` operator to make the argument-checking code more concise:

```
public string Name
{
    get => name;
    set => name = value ?? throw new ArgumentNullException(nameof(value), "Name cannot be null");
}
```

The preceding example also demonstrates how to use [expression-bodied members](#) to define a property.

- Beginning with C# 8.0, you can use the `??=` operator to replace the code of the form

```
if (variable is null)
{
    variable = expression;
}
```

with the following code:

```
variable ??= expression;
```

Operator overloadability

The operators `??` and `??=` cannot be overloaded.

C# language specification

For more information about the `??` operator, see [The null coalescing operator](#) section of the [C# language specification](#).

For more information about the `??=` operator, see the [feature proposal note](#).

See also

- [C# reference](#)

- C# operators and expressions
- ?. and ?[] operators
- ?:: operator

=> operator (C# reference)

12/28/2021 • 2 minutes to read • [Edit Online](#)

The `=>` token is supported in two forms: as the [lambda operator](#) and as a separator of a member name and the member implementation in an [expression body definition](#).

Lambda operator

In [lambda expressions](#), the lambda operator `=>` separates the input parameters on the left side from the lambda body on the right side.

The following example uses the [LINQ](#) feature with method syntax to demonstrate the usage of lambda expressions:

```
string[] words = { "bot", "apple", "apricot" };
int minLength = words
    .Where(w => w.StartsWith("a"))
    .Min(w => w.Length);
Console.WriteLine(minLength);    // output: 5

int[] numbers = { 4, 7, 10 };
int product = numbers.Aggregate(1, (interim, next) => interim * next);
Console.WriteLine(product);    // output: 280
```

Input parameters of a lambda expression are strongly typed at compile time. When the compiler can infer the types of input parameters, like in the preceding example, you may omit type declarations. If you need to specify the type of input parameters, you must do that for each parameter, as the following example shows:

```
int[] numbers = { 4, 7, 10 };
int product = numbers.Aggregate(1, (int interim, int next) => interim * next);
Console.WriteLine(product);    // output: 280
```

The following example shows how to define a lambda expression without input parameters:

```
Func<string> greet = () => "Hello, World!";
Console.WriteLine(greet());
```

For more information, see [Lambda expressions](#).

Expression body definition

An expression body definition has the following general syntax:

```
member => expression;
```

where `expression` is a valid expression. The return type of `expression` must be implicitly convertible to the member's return type. If the member:

- Has a `void` return type or
- Is a:

- Constructor
- Finalizer
- Property or indexer `set` accessor

`expression` must be a [statement expression](#). Because the expression's result is discarded, the return type of that expression can be any type.

The following example shows an expression body definition for a `Person.ToString` method:

```
public override string ToString() => $"{fname} {lname}".Trim();
```

It's a shorthand version of the following method definition:

```
public override string ToString()
{
    return $"{fname} {lname}".Trim();
}
```

Expression body definitions for methods, operators, and read-only properties are supported beginning with C# 6. Expression body definitions for constructors, finalizers, and property and indexer accessors are supported beginning with C# 7.0.

For more information, see [Expression-bodied members](#).

Operator overloadability

The `=>` operator cannot be overloaded.

C# language specification

For more information about the lambda operator, see the [Anonymous function expressions](#) section of the [C# language specification](#).

See also

- [C# reference](#)
- [C# operators and expressions](#)

:: operator (C# reference)

12/28/2021 • 2 minutes to read • [Edit Online](#)

Use the namespace alias qualifier `::` to access a member of an aliased namespace. You can use the `::` qualifier only between two identifiers. The left-hand identifier can be any of the following aliases:

- A namespace alias created with a [using alias directive](#):

```
using forwinforms = System.Drawing;
using forwpf = System.Windows;

public class Converters
{
    public static forwpf::Point Convert(forwinforms::Point point) => new forwpf::Point(point.X,
point.Y);
}
```

- An [extern alias](#).
- The `global` alias, which is the global namespace alias. The global namespace is the namespace that contains namespaces and types that are not declared inside a named namespace. When used with the `::` qualifier, the `global` alias always references the global namespace, even if there is the user-defined `global` namespace alias.

The following example uses the `global` alias to access the .NET [System](#) namespace, which is a member of the global namespace. Without the `global` alias, the user-defined `System` namespace, which is a member of the `MyCompany.MyProduct` namespace, would be accessed:

```
namespace MyCompany.MyProduct.System
{
    class Program
    {
        static void Main() => global::System.Console.WriteLine("Using global alias");
    }

    class Console
    {
        string Suggestion => "Consider renaming this class";
    }
}
```

NOTE

The `global` keyword is the global namespace alias only when it's the left-hand identifier of the `::` qualifier.

You can also use the `.` [token](#) to access a member of an aliased namespace. However, the `.` token is also used to access a type member. The `::` qualifier ensures that its left-hand identifier always references a namespace alias, even if there exists a type or namespace with the same name.

C# language specification

For more information, see the [Namespace alias qualifiers](#) section of the [C# language specification](#).

See also

- [C# reference](#)
- [C# operators and expressions](#)

await operator (C# reference)

12/28/2021 • 3 minutes to read • [Edit Online](#)

The `await` operator suspends evaluation of the enclosing `async` method until the asynchronous operation represented by its operand completes. When the asynchronous operation completes, the `await` operator returns the result of the operation, if any. When the `await` operator is applied to the operand that represents an already completed operation, it returns the result of the operation immediately without suspension of the enclosing method. The `await` operator doesn't block the thread that evaluates the `async` method. When the `await` operator suspends the enclosing `async` method, the control returns to the caller of the method.

In the following example, the `HttpClient.GetByteArrayAsync` method returns the `Task<byte[]>` instance, which represents an asynchronous operation that produces a byte array when it completes. Until the operation completes, the `await` operator suspends the `DownloadDocsMainPageAsync` method. When `DownloadDocsMainPageAsync` gets suspended, control is returned to the `Main` method, which is the caller of `DownloadDocsMainPageAsync`. The `Main` method executes until it needs the result of the asynchronous operation performed by the `DownloadDocsMainPageAsync` method. When `GetByteArrayAsync` gets all the bytes, the rest of the `DownloadDocsMainPageAsync` method is evaluated. After that, the rest of the `Main` method is evaluated.

```
using System;
using System.Net.Http;
using System.Threading.Tasks;

public class AwaitOperator
{
    public static async Task Main()
    {
        Task<int> downloading = DownloadDocsMainPageAsync();
        Console.WriteLine($"{nameof(Main)}: Launched downloading.");

        int bytesLoaded = await downloading;
        Console.WriteLine($"{nameof(Main)}: Downloaded {bytesLoaded} bytes.");
    }

    private static async Task<int> DownloadDocsMainPageAsync()
    {
        Console.WriteLine($"{nameof(DownloadDocsMainPageAsync)}: About to start downloading.");

        var client = new HttpClient();
        byte[] content = await client.GetByteArrayAsync("https://docs.microsoft.com/en-us/");

        Console.WriteLine($"{nameof(DownloadDocsMainPageAsync)}: Finished downloading.");
        return content.Length;
    }
}

// Output similar to:
// DownloadDocsMainPageAsync: About to start downloading.
// Main: Launched downloading.
// DownloadDocsMainPageAsync: Finished downloading.
// Main: Downloaded 27700 bytes.
```

The preceding example uses the `async Main` method, which is possible beginning with C# 7.1. For more information, see the [await operator in the Main method](#) section.

NOTE

For an introduction to asynchronous programming, see [Asynchronous programming with async and await](#). Asynchronous programming with `async` and `await` follows the [task-based asynchronous pattern](#).

You can use the `await` operator only in a method, [lambda expression](#), or [anonymous method](#) that is modified by the `async` keyword. Within an `async` method, you can't use the `await` operator in the body of a synchronous function, inside the block of a [lock statement](#), and in an `unsafe` context.

The operand of the `await` operator is usually of one of the following .NET types: [Task](#), [Task<TResult>](#), [ValueTask](#), or [ValueTask<TResult>](#). However, any awaitable expression can be the operand of the `await` operator. For more information, see the [Awaitable expressions](#) section of the [C# language specification](#).

The type of expression `await t` is `TResult` if the type of expression `t` is [Task<TResult>](#) or [ValueTask<TResult>](#). If the type of `t` is [Task](#) or [ValueTask](#), the type of `await t` is `void`. In both cases, if `t` throws an exception, `await t` rethrows the exception. For more information about exception handling, see the [Exceptions in async methods](#) section of the [try-catch statement](#) article.

The `async` and `await` keywords are available in C# 5 and later.

Asynchronous streams and disposables

Beginning with C# 8.0, you can work with asynchronous streams and disposables.

You use the `await foreach` statement to consume an asynchronous stream of data. For more information, see the [foreach statement](#) section of the [Iteration statements](#) article and the [Asynchronous streams](#) section of the [What's new in C# 8.0](#) article.

You use the `await using` statement to work with an asynchronously disposable object, that is, an object of a type that implements an [IAsyncDisposable](#) interface. For more information, see the [Using async disposable](#) section of the [Implement a DisposeAsync method](#) article.

await operator in the Main method

Beginning with C# 7.1, the [Main method](#), which is the application entry point, can return `Task` or `Task<int>`, enabling it to be `async` so you can use the `await` operator in its body. In earlier C# versions, to ensure that the `Main` method waits for the completion of an asynchronous operation, you can retrieve the value of the [Task<TResult>.Result](#) property of the [Task<TResult>](#) instance that is returned by the corresponding `async` method. For asynchronous operations that don't produce a value, you can call the [Task.Wait](#) method. For information about how to select the language version, see [C# language versioning](#).

C# language specification

For more information, see the [Await expressions](#) section of the [C# language specification](#).

See also

- [C# reference](#)
- [C# operators and expressions](#)
- [async](#)
- [Task asynchronous programming model](#)
- [Asynchronous programming](#)
- [Async in depth](#)

- [Walkthrough: accessing the Web by using async and await](#)
- [Tutorial: Generate and consume async streams using C# 8.0 and .NET Core 3.0](#)

default value expressions (C# reference)

12/28/2021 • 2 minutes to read • [Edit Online](#)

A default value expression produces the [default value](#) of a type. There are two kinds of default value expressions: the [default operator](#) call and a [default literal](#).

You also use the `default` keyword as the default case label within a [switch statement](#).

default operator

The argument to the `default` operator must be the name of a type or a type parameter, as the following example shows:

```
Console.WriteLine(default(int)); // output: 0
Console.WriteLine(default(object) is null); // output: True

void DisplayDefaultOf<T>()
{
    var val = default(T);
    Console.WriteLine($"Default value of {typeof(T)} is {(val == null ? "null" : val.ToString())}.");
}

DisplayDefaultOf<int?>();
DisplayDefaultOf<System.Numerics.Complex>();
DisplayDefaultOf<System.Collections.Generic.List<int>>>();
// Output:
// Default value of System.Nullable`1[System.Int32] is null.
// Default value of System.Numerics.Complex is (0, 0).
// Default value of System.Collections.Generic.List`1[System.Int32] is null.
```

default literal

Beginning with C# 7.1, you can use the `default` literal to produce the default value of a type when the compiler can infer the expression type. The `default` literal expression produces the same value as the `default(T)` expression where `T` is the inferred type. You can use the `default` literal in any of the following cases:

- In the assignment or initialization of a variable.
- In the declaration of the default value for an [optional method parameter](#).
- In a method call to provide an argument value.
- In a [return statement](#) or as an expression in an [expression-bodied member](#).

The following example shows the usage of the `default` literal:


```

T[] InitializeArray<T>(int length, T initialValue = default)
{
    if (length < 0)
    {
        throw new ArgumentOutOfRangeException(nameof(length), "Array length must be nonnegative.");
    }

    var array = new T[length];
    for (var i = 0; i < length; i++)
    {
        array[i] = initialValue;
    }
    return array;
}

void Display<T>(T[] values) => Console.WriteLine($"[ {string.Join(", ", values)} ]");

Display(InitializeArray<int>(3)); // output: [ 0, 0, 0 ]
Display(InitializeArray<bool>(4, default)); // output: [ False, False, False, False ]

System.Numerics.Complex fillValue = default;
Display(InitializeArray(3, fillValue)); // output: [ (0, 0), (0, 0), (0, 0) ]

```

C# language specification

For more information, see the [Default value expressions](#) section of the [C# language specification](#).

For more information about the `default` literal, see the [feature proposal note](#).

See also

- [C# reference](#)
- [C# operators and expressions](#)
- [Default values of C# types](#)
- [Generics in .NET](#)

delegate operator (C# reference)

12/28/2021 • 2 minutes to read • [Edit Online](#)

The `delegate` operator creates an anonymous method that can be converted to a delegate type:

```
Func<int, int, int> sum = delegate (int a, int b) { return a + b; };  
Console.WriteLine(sum(3, 4)); // output: 7
```

NOTE

Beginning with C# 3, lambda expressions provide a more concise and expressive way to create an anonymous function. Use the [=> operator](#) to construct a lambda expression:

```
Func<int, int, int> sum = (a, b) => a + b;  
Console.WriteLine(sum(3, 4)); // output: 7
```

For more information about features of lambda expressions, for example, capturing outer variables, see [Lambda expressions](#).

When you use the `delegate` operator, you might omit the parameter list. If you do that, the created anonymous method can be converted to a delegate type with any list of parameters, as the following example shows:

```
Action greet = delegate { Console.WriteLine("Hello!"); };  
greet();  
  
Action<int, double> introduce = delegate { Console.WriteLine("This is world!"); };  
introduce(42, 2.7);  
  
// Output:  
// Hello!  
// This is world!
```

That's the only functionality of anonymous methods that is not supported by lambda expressions. In all other cases, a lambda expression is a preferred way to write inline code.

Beginning with C# 9.0, you can use [discards](#) to specify two or more input parameters of an anonymous method that aren't used by the method:

```
Func<int, int, int> constant = delegate (int _, int _) { return 42; };  
Console.WriteLine(constant(3, 4)); // output: 42
```

For backwards compatibility, if only a single parameter is named `_`, `_` is treated as the name of that parameter within an anonymous method.

Also beginning with C# 9.0, you can use the `static` modifier at the declaration of an anonymous method:

```
Func<int, int, int> sum = static delegate (int a, int b) { return a + b; };  
Console.WriteLine(sum(10, 4)); // output: 14
```

A static anonymous method can't capture local variables or instance state from enclosing scopes.

You also use the `delegate` keyword to declare a [delegate type](#).

C# language specification

For more information, see the [Anonymous function expressions](#) section of the [C# language specification](#).

See also

- [C# reference](#)
- [C# operators and expressions](#)
- [=> operator](#)

is operator (C# reference)

12/28/2021 • 2 minutes to read • [Edit Online](#)

The `is` operator checks if the result of an expression is compatible with a given type. For information about the type-testing `is` operator, see the [is operator](#) section of the [Type-testing and cast operators](#) article.

Beginning with C# 7.0, you can also use the `is` operator to match an expression against a pattern, as the following example shows:

```
static bool IsFirstFridayOfOctober(DateTime date) =>
    date is { Month: 10, Day: <=7, DayOfWeek: DayOfWeek.Friday };
```

In the preceding example, the `is` operator matches an expression against a [property pattern](#) (available in C# 8.0 and later) with nested [constant](#) and [relational](#) (available in C# 9.0 and later) patterns.

The `is` operator can be useful in the following scenarios:

- To check the run-time type of an expression, as the following example shows:

```
int i = 34;
object iBoxed = i;
int? jNullable = 42;
if (iBoxed is int a && jNullable is int b)
{
    Console.WriteLine(a + b); // output 76
}
```

The preceding example shows the use of a [declaration pattern](#).

- To check for `null`, as the following example shows:

```
if (input is null)
{
    return;
}
```

When you match an expression against `null`, the compiler guarantees that no user-overloaded `==` or `!=` operator is invoked.

- Beginning with C# 9.0, you can use a [negation pattern](#) to do a non-null check, as the following example shows:

```
if (result is not null)
{
    Console.WriteLine(result.ToString());
}
```

NOTE

For the complete list of patterns supported by the `is` operator, see [Patterns](#).

C# language specification

For more information, see [The is operator](#) section of the [C# language specification](#) and the following C# language proposals:

- [Pattern matching](#)
- [Pattern matching with generics](#)

See also

- [C# reference](#)
- [C# operators and expressions](#)
- [Patterns](#)
- [Tutorial: Use pattern matching to build type-driven and data-driven algorithms](#)
- [Type-testing and cast operators](#)

nameof expression (C# reference)

12/28/2021 • 2 minutes to read • [Edit Online](#)

A `nameof` expression produces the name of a variable, type, or member as the string constant:

```
Console.WriteLine(nameof(System.Collections.Generic)); // output: Generic
Console.WriteLine(nameof(List<int>)); // output: List
Console.WriteLine(nameof(List<int>.Count)); // output: Count
Console.WriteLine(nameof(List<int>.Add)); // output: Add

var numbers = new List<int> { 1, 2, 3 };
Console.WriteLine(nameof(numbers)); // output: numbers
Console.WriteLine(nameof(numbers.Count)); // output: Count
Console.WriteLine(nameof(numbers.Add)); // output: Add
```

As the preceding example shows, in the case of a type and a namespace, the produced name is not [fully qualified](#).

In the case of [verbatim identifiers](#), the `@` character is not the part of a name, as the following example shows:

```
var @new = 5;
Console.WriteLine(nameof(@new)); // output: new
```

A `nameof` expression is evaluated at compile time and has no effect at run time.

You can use a `nameof` expression to make the argument-checking code more maintainable:

```
public string Name
{
    get => name;
    set => name = value ?? throw new ArgumentNullException(nameof(value), $"{nameof(Name)} cannot be null");
}
```

A `nameof` expression is available in C# 6 and later.

C# language specification

For more information, see the [Nameof expressions](#) section of the [C# language specification](#).

See also

- [C# reference](#)
- [C# operators and expressions](#)
- [CallerArgumentExpression](#)

new operator (C# reference)

12/28/2021 • 2 minutes to read • [Edit Online](#)

The `new` operator creates a new instance of a type.

You can also use the `new` keyword as a [member declaration modifier](#) or a [generic type constraint](#).

Constructor invocation

To create a new instance of a type, you typically invoke one of the [constructors](#) of that type using the `new` operator:

```
var dict = new Dictionary<string, int>();
dict["first"] = 10;
dict["second"] = 20;
dict["third"] = 30;

Console.WriteLine(string.Join("; ", dict.Select(entry => $"{entry.Key}: {entry.Value}")));
// Output:
// first: 10; second: 20; third: 30
```

You can use an [object or collection initializer](#) with the `new` operator to instantiate and initialize an object in one statement, as the following example shows:

```
var dict = new Dictionary<string, int>
{
    ["first"] = 10,
    ["second"] = 20,
    ["third"] = 30
};

Console.WriteLine(string.Join("; ", dict.Select(entry => $"{entry.Key}: {entry.Value}")));
// Output:
// first: 10; second: 20; third: 30
```

Beginning with C# 9.0, constructor invocation expressions are target-typed. That is, if a target type of an expression is known, you can omit a type name, as the following example shows:

```
List<int> xs = new();
List<int> ys = new(capacity: 10_000);
List<int> zs = new() { Capacity = 20_000 };

Dictionary<int, List<int>> lookup = new()
{
    [1] = new() { 1, 2, 3 },
    [2] = new() { 5, 8, 3 },
    [5] = new() { 1, 0, 4 }
};
```

As the preceding example shows, you always use parentheses in a target-typed `new` expression.

If a target type of a `new` expression is unknown (for example, when you use the `var` keyword), you must specify a type name.

Array creation

You also use the `new` operator to create an array instance, as the following example shows:

```
var numbers = new int[3];
numbers[0] = 10;
numbers[1] = 20;
numbers[2] = 30;

Console.WriteLine(string.Join(", ", numbers));
// Output:
// 10, 20, 30
```

Use array initialization syntax to create an array instance and populate it with elements in one statement. The following example shows various ways how you can do that:

```
var a = new int[3] { 10, 20, 30 };
var b = new int[] { 10, 20, 30 };
var c = new[] { 10, 20, 30 };
Console.WriteLine(c.GetType()); // output: System.Int32[]
```

For more information about arrays, see [Arrays](#).

Instantiation of anonymous types

To create an instance of an [anonymous type](#), use the `new` operator and object initializer syntax:

```
var example = new { Greeting = "Hello", Name = "World" };
Console.WriteLine($"{example.Greeting}, {example.Name}!");
// Output:
// Hello, World!
```

Destruction of type instances

You don't have to destroy earlier created type instances. Instances of both reference and value types are destroyed automatically. Instances of value types are destroyed as soon as the context that contains them is destroyed. Instances of reference types are destroyed by the [garbage collector](#) at some unspecified time after the last reference to them is removed.

For type instances that contain unmanaged resources, for example, a file handle, it's recommended to employ deterministic clean-up to ensure that the resources they contain are released as soon as possible. For more information, see the [System.IDisposable](#) API reference and the [using statement](#) article.

Operator overloadability

A user-defined type cannot overload the `new` operator.

C# language specification

For more information, see [The new operator](#) section of the [C# language specification](#).

For more information about a target-typed `new` expression, see the [feature proposal note](#).

See also

- [C# reference](#)
- [C# operators and expressions](#)
- [Object and collection initializers](#)

sizeof operator (C# reference)

12/28/2021 • 2 minutes to read • [Edit Online](#)

The `sizeof` operator returns the number of bytes occupied by a variable of a given type. The argument to the `sizeof` operator must be the name of an [unmanaged type](#) or a type parameter that is [constrained](#) to be an unmanaged type.

The `sizeof` operator requires an [unsafe](#) context. However, the expressions presented in the following table are evaluated in compile time to the corresponding constant values and don't require an unsafe context:

EXPRESSION	CONSTANT VALUE
<code>sizeof(sbyte)</code>	1
<code>sizeof(byte)</code>	1
<code>sizeof(short)</code>	2
<code>sizeof(ushort)</code>	2
<code>sizeof(int)</code>	4
<code>sizeof(uint)</code>	4
<code>sizeof(long)</code>	8
<code>sizeof(ulong)</code>	8
<code>sizeof(char)</code>	2
<code>sizeof(float)</code>	4
<code>sizeof(double)</code>	8
<code>sizeof(decimal)</code>	16
<code>sizeof(bool)</code>	1

You also don't need to use an unsafe context when the operand of the `sizeof` operator is the name of an [enum](#) type.

The following example demonstrates the usage of the `sizeof` operator:

```

using System;

public struct Point
{
    public Point(byte tag, double x, double y) => (Tag, X, Y) = (tag, x, y);

    public byte Tag { get; }
    public double X { get; }
    public double Y { get; }
}

public class SizeOfOperator
{
    public static void Main()
    {
        Console.WriteLine(sizeof(byte)); // output: 1
        Console.WriteLine(sizeof(double)); // output: 8

        DisplaySizeOf<Point>(); // output: Size of Point is 24
        DisplaySizeOf<decimal>(); // output: Size of System.Decimal is 16

        unsafe
        {
            Console.WriteLine(sizeof(Point*)); // output: 8
        }

        static unsafe void DisplaySizeOf<T>() where T : unmanaged
        {
            Console.WriteLine($"Size of {typeof(T)} is {sizeof(T)}");
        }
    }
}

```

The `sizeof` operator returns a number of bytes that would be allocated by the common language runtime in managed memory. For [struct](#) types, that value includes any padding, as the preceding example demonstrates. The result of the `sizeof` operator might differ from the result of the [Marshal.SizeOf](#) method, which returns the size of a type in *unmanaged* memory.

C# language specification

For more information, see [The sizeof operator](#) section of the [C# language specification](#).

See also

- [C# reference](#)
- [C# operators and expressions](#)
- [Pointer related operators](#)
- [Pointer types](#)
- [Memory and span-related types](#)
- [Generics in .NET](#)

stackalloc expression (C# reference)

12/28/2021 • 3 minutes to read • [Edit Online](#)

A `stackalloc` expression allocates a block of memory on the stack. A stack allocated memory block created during the method execution is automatically discarded when that method returns. You cannot explicitly free the memory allocated with `stackalloc`. A stack allocated memory block is not subject to [garbage collection](#) and doesn't have to be pinned with a `fixed` statement.

You can assign the result of a `stackalloc` expression to a variable of one of the following types:

- Beginning with C# 7.2, [System.Span<T>](#) or [System.ReadOnlySpan<T>](#), as the following example shows:

```
int length = 3;
Span<int> numbers = stackalloc int[length];
for (var i = 0; i < length; i++)
{
    numbers[i] = i;
}
```

You don't have to use an [unsafe](#) context when you assign a stack allocated memory block to a [Span<T>](#) or [ReadOnlySpan<T>](#) variable.

When you work with those types, you can use a `stackalloc` expression in [conditional](#) or assignment expressions, as the following example shows:

```
int length = 1000;
Span<byte> buffer = length <= 1024 ? stackalloc byte[length] : new byte[length];
```

Beginning with C# 8.0, you can use a `stackalloc` expression inside other expressions whenever a [Span<T>](#) or [ReadOnlySpan<T>](#) variable is allowed, as the following example shows:

```
Span<int> numbers = stackalloc[] { 1, 2, 3, 4, 5, 6 };
var ind = numbers.IndexOfAny(stackalloc[] { 2, 4, 6, 8 });
Console.WriteLine(ind); // output: 1
```

NOTE

We recommend using [Span<T>](#) or [ReadOnlySpan<T>](#) types to work with stack allocated memory whenever possible.

- A [pointer type](#), as the following example shows:

```
unsafe
{
    int length = 3;
    int* numbers = stackalloc int[length];
    for (var i = 0; i < length; i++)
    {
        numbers[i] = i;
    }
}
```

As the preceding example shows, you must use an `unsafe` context when you work with pointer types.

In the case of pointer types, you can use a `stackalloc` expression only in a local variable declaration to initialize the variable.

The amount of memory available on the stack is limited. If you allocate too much memory on the stack, a [StackOverflowException](#) is thrown. To avoid that, follow the rules below:

- Limit the amount of memory you allocate with `stackalloc`. For example, if the intended buffer size is below a certain limit, you allocate the memory on the stack; otherwise, use an array of the required length, as the following code shows:

```
const int MaxStackLimit = 1024;
Span<byte> buffer = inputLength <= MaxStackLimit ? stackalloc byte[inputLength] : new
byte[inputLength];
```

NOTE

Because the amount of memory available on the stack depends on the environment in which the code is executed, be conservative when you define the actual limit value.

- Avoid using `stackalloc` inside loops. Allocate the memory block outside a loop and reuse it inside the loop.

The content of the newly allocated memory is undefined. You should initialize it before the use. For example, you can use the [Span<T>.Clear](#) method that sets all the items to the default value of type `T`.

Beginning with C# 7.3, you can use array initializer syntax to define the content of the newly allocated memory. The following example demonstrates various ways to do that:

```
Span<int> first = stackalloc int[3] { 1, 2, 3 };
Span<int> second = stackalloc int[] { 1, 2, 3 };
ReadOnlySpan<int> third = stackalloc[] { 1, 2, 3 };
```

In expression `stackalloc T[E]`, `T` must be an [unmanaged type](#) and `E` must evaluate to a non-negative `int` value.

Security

The use of `stackalloc` automatically enables buffer overrun detection features in the common language runtime (CLR). If a buffer overrun is detected, the process is terminated as quickly as possible to minimize the chance that malicious code is executed.

C# language specification

For more information, see the [Stack allocation](#) section of the [C# language specification](#) and the [Permit `stackalloc` in nested contexts](#) feature proposal note.

See also

- [C# reference](#)
- [C# operators and expressions](#)
- [Pointer related operators](#)

- [Pointer types](#)
- [Memory and span-related types](#)
- [Dos and Don'ts of stackalloc](#)

switch expression (C# reference)

12/28/2021 • 3 minutes to read • [Edit Online](#)

Beginning with C# 8.0, you use the `switch` expression to evaluate a single expression from a list of candidate expressions based on a pattern match with an input expression. For information about the `switch` statement that supports `switch`-like semantics in a statement context, see the [switch statement](#) section of the [Selection statements](#) article.

The following example demonstrates a `switch` expression, which converts values of an `enum` representing visual directions in an online map to the corresponding cardinal directions:

```
public static class SwitchExample
{
    public enum Direction
    {
        Up,
        Down,
        Right,
        Left
    }

    public enum Orientation
    {
        North,
        South,
        East,
        West
    }

    public static Orientation ToOrientation(Direction direction) => direction switch
    {
        Direction.Up    => Orientation.North,
        Direction.Right => Orientation.East,
        Direction.Down  => Orientation.South,
        Direction.Left  => Orientation.West,
        _ => throw new ArgumentOutOfRangeException(nameof(direction), $"Not expected direction value: {direction}"),
    };

    public static void Main()
    {
        var direction = Direction.Right;
        Console.WriteLine($"Map view direction is {direction}");
        Console.WriteLine($"Cardinal orientation is {ToOrientation(direction)}");
        // Output:
        // Map view direction is Right
        // Cardinal orientation is East
    }
}
```

The preceding example shows the basic elements of a `switch` expression:

- An expression followed by the `switch` keyword. In the preceding example, it's the `direction` method parameter.
- The `switch` *expression arms*, separated by commas. Each `switch` expression arm contains a *pattern*, an optional *case guard*, the `=>` token, and an *expression*.

At the preceding example, a `switch` expression uses the following patterns:

- A [constant pattern](#): to handle the defined values of the `Direction` enumeration.
- A [discard pattern](#): to handle any integer value that doesn't have the corresponding member of the `Direction` enumeration (for example, `(Direction)10`). That makes the `switch` expression [exhaustive](#).

IMPORTANT

For information about the patterns supported by the `switch` expression and more examples, see [Patterns](#).

The result of a `switch` expression is the value of the expression of the first `switch` expression arm whose pattern matches the input expression and whose case guard, if present, evaluates to `true`. The `switch` expression arms are evaluated in text order.

The compiler generates an error when a lower `switch` expression arm can't be chosen because a higher `switch` expression arm matches all its values.

Case guards

A pattern may be not expressive enough to specify the condition for the evaluation of an arm's expression. In such a case, you can use a case guard. That is an additional condition that must be satisfied together with a matched pattern. A case guard must be a Boolean expression. You specify a case guard after the `when` keyword that follows a pattern, as the following example shows:

```
public readonly struct Point
{
    public Point(int x, int y) => (X, Y) = (x, y);

    public int X { get; }
    public int Y { get; }
}

static Point Transform(Point point) => point switch
{
    { X: 0, Y: 0 } => new Point(0, 0),
    { X: var x, Y: var y } when x < y => new Point(x + y, y),
    { X: var x, Y: var y } when x > y => new Point(x - y, y),
    { X: var x, Y: var y } => new Point(2 * x, 2 * y),
};
```

The preceding example uses [property patterns](#) with nested [var patterns](#).

Non-exhaustive switch expressions

If none of a `switch` expression's patterns matches an input value, the runtime throws an exception. In .NET Core 3.0 and later versions, the exception is a [System.Runtime.CompilerServices.SwitchExpressionException](#). In .NET Framework, the exception is an [InvalidOperationException](#). The compiler generates a warning if a `switch` expression doesn't handle all possible input values.

TIP

To guarantee that a `switch` expression handles all possible input values, provide a `switch` expression arm with a [discard pattern](#).

C# language specification

For more information, see the [switch expression](#) section of the [feature proposal note](#).

See also

- [C# reference](#)
- [C# operators and expressions](#)
- [Patterns](#)
- [Tutorial: Use pattern matching to build type-driven and data-driven algorithms](#)
- [switch statement](#)

true and false operators (C# reference)

12/28/2021 • 2 minutes to read • [Edit Online](#)

The `true` operator returns the `bool` value `true` to indicate that its operand is definitely true. The `false` operator returns the `bool` value `false` to indicate that its operand is definitely false. The `true` and `false` operators are not guaranteed to complement each other. That is, both the `true` and `false` operator might return the `bool` value `false` for the same operand. If a type defines one of the two operators, it must also define another operator.

TIP

Use the `bool?` type, if you need to support the three-valued logic (for example, when you work with databases that support a three-valued Boolean type). C# provides the `&` and `|` operators that support the three-valued logic with the `bool?` operands. For more information, see the [Nullable Boolean logical operators](#) section of the [Boolean logical operators](#) article.

Boolean expressions

A type with the defined `true` operator can be the type of a result of a controlling conditional expression in the `if`, `do`, `while`, and `for` statements and in the [conditional operator](#) `?:`. For more information, see the [Boolean expressions](#) section of the [C# language specification](#).

User-defined conditional logical operators

If a type with the defined `true` and `false` operators [overloads](#) the [logical OR operator](#) `|` or the [logical AND operator](#) `&` in a certain way, the [conditional logical OR operator](#) `||` or [conditional logical AND operator](#) `&&`, respectively, can be evaluated for the operands of that type. For more information, see the [User-defined conditional logical operators](#) section of the [C# language specification](#).

Example

The following example presents the type that defines both `true` and `false` operators. The type also overloads the logical AND operator `&` in such a way that the `&&` operator also can be evaluated for the operands of that type.

```

using System;

public struct LaunchStatus
{
    public static readonly LaunchStatus Green = new LaunchStatus(0);
    public static readonly LaunchStatus Yellow = new LaunchStatus(1);
    public static readonly LaunchStatus Red = new LaunchStatus(2);

    private int status;

    private LaunchStatus(int status)
    {
        this.status = status;
    }

    public static bool operator true(LaunchStatus x) => x == Green || x == Yellow;
    public static bool operator false(LaunchStatus x) => x == Red;

    public static LaunchStatus operator &(LaunchStatus x, LaunchStatus y)
    {
        if (x == Red || y == Red || (x == Yellow && y == Yellow))
        {
            return Red;
        }

        if (x == Yellow || y == Yellow)
        {
            return Yellow;
        }

        return Green;
    }

    public static bool operator ==(LaunchStatus x, LaunchStatus y) => x.status == y.status;
    public static bool operator !=(LaunchStatus x, LaunchStatus y) => !(x == y);

    public override bool Equals(object obj) => obj is LaunchStatus other && this == other;
    public override int GetHashCode() => status;
}

public class LaunchStatusTest
{
    public static void Main()
    {
        LaunchStatus okToLaunch = GetFuelLaunchStatus() && GetNavigationLaunchStatus();
        Console.WriteLine(okToLaunch ? "Ready to go!" : "Wait!");
    }

    static LaunchStatus GetFuelLaunchStatus()
    {
        Console.WriteLine("Getting fuel launch status...");
        return LaunchStatus.Red;
    }

    static LaunchStatus GetNavigationLaunchStatus()
    {
        Console.WriteLine("Getting navigation launch status...");
        return LaunchStatus.Yellow;
    }
}

```

Notice the short-circuiting behavior of the `&&` operator. When the `GetFuelLaunchStatus` method returns `LaunchStatus.Red`, the right-hand operand of the `&&` operator is not evaluated. That is because `LaunchStatus.Red` is definitely false. Then the result of the logical AND doesn't depend on the value of the right-hand operand. The output of the example is as follows:

```
Getting fuel launch status...  
Wait!
```

See also

- [C# reference](#)
- [C# operators and expressions](#)

with expression (C# reference)

12/28/2021 • 3 minutes to read • [Edit Online](#)

Available in C# 9.0 and later, a `with` expression produces a copy of its operand with the specified properties and fields modified:

```
using System;

public class WithExpressionBasicExample
{
    public record NamedPoint(string Name, int X, int Y);

    public static void Main()
    {
        var p1 = new NamedPoint("A", 0, 0);
        Console.WriteLine($"{nameof(p1)}: {p1}"); // output: p1: NamedPoint { Name = A, X = 0, Y = 0 }

        var p2 = p1 with { Name = "B", X = 5 };
        Console.WriteLine($"{nameof(p2)}: {p2}"); // output: p2: NamedPoint { Name = B, X = 5, Y = 0 }

        var p3 = p1 with
        {
            Name = "C",
            Y = 4
        };
        Console.WriteLine($"{nameof(p3)}: {p3}"); // output: p3: NamedPoint { Name = C, X = 0, Y = 4 }

        Console.WriteLine($"{nameof(p1)}: {p1}"); // output: p1: NamedPoint { Name = A, X = 0, Y = 0 }

        var apples = new { Item = "Apples", Price = "1.19" };
        Console.WriteLine($"original apples: {apples}");
        var saleApples = apples with { Price = "0.79" };
        Console.WriteLine($"sale apples: {saleApples}");
    }
}
```

As the preceding example shows, you use [object initializer](#) syntax to specify what members to modify and their new values.

In C# 9.0, a left-hand operand of a `with` expression must be of a [record type](#). Beginning with C# 10, a left-hand operand of a `with` expression can also be of a [structure type](#) or an [anonymous type](#).

The result of a `with` expression has the same run-time type as the expression's operand, as the following example shows:

```

using System;

public class InheritanceExample
{
    public record Point(int X, int Y);
    public record NamedPoint(string Name, int X, int Y) : Point(X, Y);

    public static void Main()
    {
        Point p1 = new NamedPoint("A", 0, 0);
        Point p2 = p1 with { X = 5, Y = 3 };
        Console.WriteLine(p2 is NamedPoint); // output: True
        Console.WriteLine(p2); // output: NamedPoint { X = 5, Y = 3, Name = A }
    }
}

```

In the case of a reference-type member, only the reference to a member instance is copied when an operand is copied. Both the copy and original operand have access to the same reference-type instance. The following example demonstrates that behavior:

```

using System;
using System.Collections.Generic;

public class ExampleWithReferenceType
{
    public record TaggedNumber(int Number, List<string> Tags)
    {
        public string PrintTags() => string.Join(", ", Tags);
    }

    public static void Main()
    {
        var original = new TaggedNumber(1, new List<string> { "A", "B" });

        var copy = original with { Number = 2 };
        Console.WriteLine($"Tags of {nameof(copy)}: {copy.PrintTags()}");
        // output: Tags of copy: A, B

        original.Tags.Add("C");
        Console.WriteLine($"Tags of {nameof(copy)}: {copy.PrintTags()}");
        // output: Tags of copy: A, B, C
    }
}

```

Custom copy semantics

Any record class type has the *copy constructor*. That is a constructor with a single parameter of the containing record type. It copies the state of its argument to a new record instance. At evaluation of a `with` expression, the copy constructor gets called to instantiate a new record instance based on an original record. After that, the new instance gets updated according to the specified modifications. By default, the copy constructor is implicit, that is, compiler-generated. If you need to customize the record copy semantics, explicitly declare a copy constructor with the desired behavior. The following example updates the preceding example with an explicit copy constructor. The new copy behavior is to copy list items instead of a list reference when a record is copied:

```

using System;
using System.Collections.Generic;

public class UserDefinedCopyConstructorExample
{
    public record TaggedNumber(int Number, List<string> Tags)
    {
        protected TaggedNumber(TaggedNumber original)
        {
            Number = original.Number;
            Tags = new List<string>(original.Tags);
        }

        public string PrintTags() => string.Join(", ", Tags);
    }

    public static void Main()
    {
        var original = new TaggedNumber(1, new List<string> { "A", "B" });

        var copy = original with { Number = 2 };
        Console.WriteLine($"Tags of {nameof(copy)}: {copy.PrintTags()}");
        // output: Tags of copy: A, B

        original.Tags.Add("C");
        Console.WriteLine($"Tags of {nameof(copy)}: {copy.PrintTags()}");
        // output: Tags of copy: A, B
    }
}

```

You cannot customize the copy semantics for structure types.

C# language specification

For more information, see the following sections of the [records feature proposal note](#):

- [with expression](#)
- [Copy and Clone members](#)

See also

- [C# reference](#)
- [C# operators and expressions](#)
- [Records](#)
- [Structure types](#)

Operator overloading (C# reference)

12/28/2021 • 3 minutes to read • [Edit Online](#)

A user-defined type can overload a predefined C# operator. That is, a type can provide the custom implementation of an operation in case one or both of the operands are of that type. The [Overloadable operators](#) section shows which C# operators can be overloaded.

Use the `operator` keyword to declare an operator. An operator declaration must satisfy the following rules:

- It includes both a `public` and a `static` modifier.
- A unary operator has one input parameter. A binary operator has two input parameters. In each case, at least one parameter must have type `T` or `T?` where `T` is the type that contains the operator declaration.

The following example defines a simplified structure to represent a rational number. The structure overloads some of the [arithmetic operators](#):


```

using System;

public readonly struct Fraction
{
    private readonly int num;
    private readonly int den;

    public Fraction(int numerator, int denominator)
    {
        if (denominator == 0)
        {
            throw new ArgumentException("Denominator cannot be zero.", nameof(denominator));
        }
        num = numerator;
        den = denominator;
    }

    public static Fraction operator +(Fraction a) => a;
    public static Fraction operator -(Fraction a) => new Fraction(-a.num, a.den);

    public static Fraction operator +(Fraction a, Fraction b)
        => new Fraction(a.num * b.den + b.num * a.den, a.den * b.den);

    public static Fraction operator -(Fraction a, Fraction b)
        => a + (-b);

    public static Fraction operator *(Fraction a, Fraction b)
        => new Fraction(a.num * b.num, a.den * b.den);

    public static Fraction operator /(Fraction a, Fraction b)
    {
        if (b.num == 0)
        {
            throw new DivideByZeroException();
        }
        return new Fraction(a.num * b.den, a.den * b.num);
    }

    public override string ToString() => $"{num} / {den}";
}

public static class OperatorOverloading
{
    public static void Main()
    {
        var a = new Fraction(5, 4);
        var b = new Fraction(1, 2);
        Console.WriteLine(-a);    // output: -5 / 4
        Console.WriteLine(a + b); // output: 14 / 8
        Console.WriteLine(a - b); // output: 6 / 8
        Console.WriteLine(a * b); // output: 5 / 8
        Console.WriteLine(a / b); // output: 10 / 4
    }
}

```

You could extend the preceding example by [defining an implicit conversion](#) from `int` to `Fraction`. Then, overloaded operators would support arguments of those two types. That is, it would become possible to add an integer to a fraction and obtain a fraction as a result.

You also use the `operator` keyword to define a custom type conversion. For more information, see [User-defined conversion operators](#).

Overloadable operators

The following table provides information about overloadability of C# operators:

OPERATORS	OVERLOADABILITY
<code>+x</code> , <code>-x</code> , <code>!x</code> , <code>~x</code> , <code>++</code> , <code>--</code> , <code>true</code> , <code>false</code>	These unary operators can be overloaded.
<code>x + y</code> , <code>x - y</code> , <code>x * y</code> , <code>x / y</code> , <code>x % y</code> , <code>x & y</code> , <code>x y</code> , <code>x ^ y</code> , <code>x << y</code> , <code>x >> y</code> , <code>x == y</code> , <code>x != y</code> , <code>x < y</code> , <code>x > y</code> , <code>x <= y</code> , <code>x >= y</code>	These binary operators can be overloaded. Certain operators must be overloaded in pairs; for more information, see the note that follows this table.
<code>x && y</code> , <code>x y</code>	Conditional logical operators cannot be overloaded. However, if a type with the overloaded <code>true</code> and <code>false</code> operators also overloads the <code>&</code> or <code> </code> operator in a certain way, the <code>&&</code> or <code> </code> operator, respectively, can be evaluated for the operands of that type. For more information, see the User-defined conditional logical operators section of the C# language specification .
<code>a[i]</code> , <code>a?[i]</code>	Element access is not considered an overloadable operator, but you can define an indexer .
<code>(T)x</code>	The cast operator cannot be overloaded, but you can define custom type conversions that can be performed by a cast expression. For more information, see User-defined conversion operators .
<code>+=</code> , <code>-=</code> , <code>*=</code> , <code>/=</code> , <code>%=</code> , <code>&=</code> , <code> =</code> , <code>^=</code> , <code><=<</code> , <code>>>=</code>	Compound assignment operators cannot be explicitly overloaded. However, when you overload a binary operator, the corresponding compound assignment operator, if any, is also implicitly overloaded. For example, <code>+=</code> is evaluated using <code>+</code> , which can be overloaded.
<code>^x</code> , <code>x = y</code> , <code>x.y</code> , <code>x?.y</code> , <code>c ? t : f</code> , <code>x ?? y</code> , <code>x ??= y</code> , <code>x.y</code> , <code>x->y</code> , <code>=></code> , <code>f(x)</code> , <code>as</code> , <code>await</code> , <code>checked</code> , <code>unchecked</code> , <code>default</code> , <code>delegate</code> , <code>is</code> , <code>nameof</code> , <code>new</code> , <code>sizeof</code> , <code>stackalloc</code> , <code>switch</code> , <code>typeof</code> , <code>with</code>	These operators cannot be overloaded.

NOTE

The comparison operators must be overloaded in pairs. That is, if either operator of a pair is overloaded, the other operator must be overloaded as well. Such pairs are as follows:

- `==` and `!=` operators
- `<` and `>` operators
- `<=` and `>=` operators

C# language specification

For more information, see the following sections of the [C# language specification](#):

- [Operator overloading](#)
- [Operators](#)

See also

- [C# reference](#)
- [C# operators and expressions](#)

- [User-defined conversion operators](#)
- [Design guidelines - Operator overloads](#)
- [Design guidelines - Equality operators](#)
- [Why are overloaded operators always static in C#?](#)

Iteration statements (C# reference)

12/28/2021 • 7 minutes to read • [Edit Online](#)

The following statements repeatedly execute a statement or a block of statements:

- The `for` statement: executes its body while a specified Boolean expression evaluates to `true`.
- The `foreach` statement: enumerates the elements of a collection and executes its body for each element of the collection.
- The `do` statement: conditionally executes its body one or more times.
- The `while` statement: conditionally executes its body zero or more times.

At any point within the body of an iteration statement, you can break out of the loop by using the `break` statement, or step to the next iteration in the loop by using the `continue` statement.

The `for` statement

The `for` statement executes a statement or a block of statements while a specified Boolean expression evaluates to `true`. The following example shows the `for` statement that executes its body while an integer counter is less than three:

```
for (int i = 0; i < 3; i++)
{
    Console.Write(i);
}
// Output:
// 012
```

The preceding example shows the elements of the `for` statement:

- The *initializer* section that is executed only once, before entering the loop. Typically, you declare and initialize a local loop variable in that section. The declared variable can't be accessed from outside the `for` statement.

The *initializer* section in the preceding example declares and initializes an integer counter variable:

```
int i = 0
```

- The *condition* section that determines if the next iteration in the loop should be executed. If it evaluates to `true` or is not present, the next iteration is executed; otherwise, the loop is exited. The *condition* section must be a Boolean expression.

The *condition* section in the preceding example checks if a counter value is less than three:

```
i < 3
```

- The *iterator* section that defines what happens after each execution of the body of the loop.

The *iterator* section in the preceding example increments the counter:

```
i++
```

- The body of the loop, which must be a statement or a block of statements.

The iterator section can contain zero or more of the following statement expressions, separated by commas:

- prefix or postfix [increment](#) expression, such as `++i` or `i++`
- prefix or postfix [decrement](#) expression, such as `--i` or `i--`
- [assignment](#)
- invocation of a method
- [await](#) expression
- creation of an object by using the [new](#) operator

If you don't declare a loop variable in the initializer section, you can use zero or more of the expressions from the preceding list in the initializer section as well. The following example shows several less common usages of the initializer and iterator sections: assigning a value to an external variable in the initializer section, invoking a method in both the initializer and the iterator sections, and changing the values of two variables in the iterator section:

```
int i;
int j = 3;
for (i = 0, Console.WriteLine($"Start: i={i}, j={j}"); i < j; i++, j--, Console.WriteLine($"Step: i={i}, j={j}"))
{
    //...
}
// Output:
// Start: i=0, j=3
// Step: i=1, j=2
// Step: i=2, j=1
```

All the sections of the `for` statement are optional. For example, the following code defines the infinite `for` loop:

```
for ( ; ; )
{
    //...
}
```

The `foreach` statement

The `foreach` statement executes a statement or a block of statements for each element in an instance of the type that implements the [System.Collections.IEnumerable](#) or [System.Collections.Generic.IEnumerable<T>](#) interface, as the following example shows:

```
var fibNumbers = new List<int> { 0, 1, 1, 2, 3, 5, 8, 13 };
foreach (int element in fibNumbers)
{
    Console.Write($"{element} ");
}
// Output:
// 0 1 1 2 3 5 8 13
```

The `foreach` statement isn't limited to those types. You can use it with an instance of any type that satisfies the

following conditions:

- A type has the public parameterless `GetEnumerator` method. Beginning with C# 9.0, the `GetEnumerator` method can be a type's [extension method](#).
- The return type of the `GetEnumerator` method has the public `Current` property and the public parameterless `MoveNext` method whose return type is `bool`.

The following example uses the `foreach` statement with an instance of the `System.Span<T>` type, which doesn't implement any interfaces:

```
Span<int> numbers = new int[] { 3, 14, 15, 92, 6 };
foreach (int number in numbers)
{
    Console.WriteLine($"{number} ");
}
// Output:
// 3 14 15 92 6
```

Beginning with C# 7.3, if the enumerator's `Current` property returns a [reference return value](#) (`ref T` where `T` is the type of a collection element), you can declare an iteration variable with the `ref` or `ref readonly` modifier, as the following example shows:

```
Span<int> storage = stackalloc int[10];
int num = 0;
foreach (ref int item in storage)
{
    item = num++;
}
foreach (ref readonly var item in storage)
{
    Console.WriteLine($"{item} ");
}
// Output:
// 0 1 2 3 4 5 6 7 8 9
```

If the `foreach` statement is applied to `null`, a [NullReferenceException](#) is thrown. If the source collection of the `foreach` statement is empty, the body of the `foreach` statement isn't executed and skipped.

await foreach

Beginning with C# 8.0, you can use the `await foreach` statement to consume an asynchronous stream of data, that is, the collection type that implements the [IAsyncEnumerable<T>](#) interface. Each iteration of the loop may be suspended while the next element is retrieved asynchronously. The following example shows how to use the `await foreach` statement:

```
await foreach (var item in GenerateSequenceAsync())
{
    Console.WriteLine(item);
}
```

You can also use the `await foreach` statement with an instance of any type that satisfies the following conditions:

- A type has the public parameterless `GetAsyncEnumerator` method. That method can be a type's [extension method](#).
- The return type of the `GetAsyncEnumerator` method has the public `Current` property and the public parameterless `MoveNextAsync` method whose return type is `Task<bool>`, `ValueTask<bool>`, or any other

awaitable type whose `awaiter`'s `GetResult` method returns a `bool` value.

By default, stream elements are processed in the captured context. If you want to disable capturing of the context, use the [TaskAsyncEnumerableExtensions.ConfigureAwait](#) extension method. For more information about synchronization contexts and capturing the current context, see [Consuming the Task-based asynchronous pattern](#). For more information about asynchronous streams, see the [Asynchronous streams](#) section of the [What's new in C# 8.0](#) article.

Type of an iteration variable

You can use the `var` keyword to let the compiler infer the type of an iteration variable in the `foreach` statement, as the following code shows:

```
foreach (var item in collection) { }
```

You can also explicitly specify the type of an iteration variable, as the following code shows:

```
IEnumerable<T> collection = new T[5];  
foreach (V item in collection) { }
```

In the preceding form, type `T` of a collection element must be implicitly or explicitly convertible to type `V` of an iteration variable. If an explicit conversion from `T` to `V` fails at run time, the `foreach` statement throws an [InvalidCastException](#). For example, if `T` is a non-sealed class type, `V` can be any interface type, even the one that `T` doesn't implement. At run time, the type of a collection element may be the one that derives from `T` and actually implements `V`. If that's not the case, an [InvalidCastException](#) is thrown.

The `do` statement

The `do` statement executes a statement or a block of statements while a specified Boolean expression evaluates to `true`. Because that expression is evaluated after each execution of the loop, a `do` loop executes one or more times. This differs from a [while](#) loop, which executes zero or more times.

The following example shows the usage of the `do` statement:

```
int n = 0;  
do  
{  
    Console.Write(n);  
    n++;  
} while (n < 5);  
// Output:  
// 01234
```

The `while` statement

The `while` statement executes a statement or a block of statements while a specified Boolean expression evaluates to `true`. Because that expression is evaluated before each execution of the loop, a `while` loop executes zero or more times. This differs from a [do](#) loop, which executes one or more times.

The following example shows the usage of the `while` statement:

```
int n = 0;
while (n < 5)
{
    Console.Write(n);
    n++;
}
// Output:
// 01234
```

C# language specification

For more information, see the following sections of the [C# language specification](#):

- [The `for` statement](#)
- [The `foreach` statement](#)
- [The `do` statement](#)
- [The `while` statement](#)

For more information about features added in C# 8.0 and later, see the following feature proposal notes:

- [Async streams \(C# 8.0\)](#)
- [Extension `GetEnumerator` support for `foreach` loops \(C# 9.0\)](#)

See also

- [C# reference](#)
- [Using foreach with arrays](#)
- [Iterators](#)

Selection statements (C# reference)

12/28/2021 • 5 minutes to read • [Edit Online](#)

The following statements select statements to execute from a number of possible statements based on the value of an expression:

- The `if statement`: selects a statement to execute based on the value of a Boolean expression.
- The `switch statement`: selects a statement list to execute based on a pattern match with an expression.

The `if` statement

An `if` statement can be any of the following two forms:

- An `if` statement with an `else` part selects one of the two statements to execute based on the value of a Boolean expression, as the following example shows:

```
DisplayWeatherReport(15.0); // Output: Cold.
DisplayWeatherReport(24.0); // Output: Perfect!

void DisplayWeatherReport(double tempInCelsius)
{
    if (tempInCelsius < 20.0)
    {
        Console.WriteLine("Cold.");
    }
    else
    {
        Console.WriteLine("Perfect!");
    }
}
```

- An `if` statement without an `else` part executes its body only if a Boolean expression evaluates to `true`, as the following example shows:

```
DisplayMeasurement(45); // Output: The measurement value is 45
DisplayMeasurement(-3); // Output: Warning: not acceptable value! The measurement value is -3

void DisplayMeasurement(double value)
{
    if (value < 0 || value > 100)
    {
        Console.Write("Warning: not acceptable value! ");
    }

    Console.WriteLine($"The measurement value is {value}");
}
```

You can nest `if` statements to check multiple conditions, as the following example shows:

```

DisplayCharacter('f'); // Output: A lowercase letter: f
DisplayCharacter('R'); // Output: An uppercase letter: R
DisplayCharacter('8'); // Output: A digit: 8
DisplayCharacter(','); // Output: Not alphanumeric character: ,

void DisplayCharacter(char ch)
{
    if (char.IsUpper(ch))
    {
        Console.WriteLine($"An uppercase letter: {ch}");
    }
    else if (char.IsLower(ch))
    {
        Console.WriteLine($"A lowercase letter: {ch}");
    }
    else if (char.IsDigit(ch))
    {
        Console.WriteLine($"A digit: {ch}");
    }
    else
    {
        Console.WriteLine($"Not alphanumeric character: {ch}");
    }
}

```

In an expression context, you can use the [conditional operator](#) `?:` to evaluate one of the two expressions based on the value of a Boolean expression.

The `switch` statement

The `switch` statement selects a statement list to execute based on a pattern match with a match expression, as the following example shows:

```

DisplayMeasurement(-4); // Output: Measured value is -4; too low.
DisplayMeasurement(5); // Output: Measured value is 5.
DisplayMeasurement(30); // Output: Measured value is 30; too high.
DisplayMeasurement(double.NaN); // Output: Failed measurement.

void DisplayMeasurement(double measurement)
{
    switch (measurement)
    {
        case < 0.0:
            Console.WriteLine($"Measured value is {measurement}; too low.");
            break;

        case > 15.0:
            Console.WriteLine($"Measured value is {measurement}; too high.");
            break;

        case double.NaN:
            Console.WriteLine("Failed measurement.");
            break;

        default:
            Console.WriteLine($"Measured value is {measurement}.");
            break;
    }
}

```

At the preceding example, the `switch` statement uses the following patterns:

- A [relational pattern](#): to compare an expression result with a constant.
- A [constant pattern](#): to test if an expression result equals a constant.

IMPORTANT

For information about the patterns supported by the `switch` statement, see [Patterns](#).

The preceding example also demonstrates the `default` case. The `default` case specifies statements to execute when a match expression doesn't match any other case pattern. If a match expression doesn't match any case pattern and there is no `default` case, control falls through a `switch` statement.

A `switch` statement executes the *statement list* in the first *switch section* whose *case pattern* matches a match expression and whose [case guard](#), if present, evaluates to `true`. A `switch` statement evaluates case patterns in text order from top to bottom. The compiler generates an error when a `switch` statement contains an unreachable case. That is a case that is already handled by an upper case or whose pattern is impossible to match.

NOTE

The `default` case can appear in any place within a `switch` statement. Regardless of its position, the `default` case is always evaluated last and only if all other case patterns aren't matched.

You can specify multiple case patterns for one section of a `switch` statement, as the following example shows:

```
DisplayMeasurement(-4); // Output: Measured value is -4; out of an acceptable range.
DisplayMeasurement(50); // Output: Measured value is 50.
DisplayMeasurement(132); // Output: Measured value is 132; out of an acceptable range.

void DisplayMeasurement(int measurement)
{
    switch (measurement)
    {
        case < 0:
        case > 100:
            Console.WriteLine($"Measured value is {measurement}; out of an acceptable range.");
            break;

        default:
            Console.WriteLine($"Measured value is {measurement}.");
            break;
    }
}
```

Within a `switch` statement, control cannot fall through from one switch section to the next. As the examples in this section show, typically you use the `break` statement at the end of each switch section to pass control out of a `switch` statement. You can also use the [return](#) and [throw](#) statements to pass control out of a `switch` statement. To imitate the fall-through behavior and pass control to other switch section, you can use the [goto statement](#).

In an expression context, you can use the [switch expression](#) to evaluate a single expression from a list of candidate expressions based on a pattern match with an expression.

Case guards

A case pattern may be not expressive enough to specify the condition for the execution of the switch section. In such a case, you can use a *case guard*. That is an additional condition that must be satisfied together with a matched pattern. A case guard must be a Boolean expression. You specify a case guard after the `when` keyword

that follows a pattern, as the following example shows:

```
DisplayMeasurements(3, 4); // Output: First measurement is 3, second measurement is 4.
DisplayMeasurements(5, 5); // Output: Both measurements are valid and equal to 5.

void DisplayMeasurements(int a, int b)
{
    switch ((a, b))
    {
        case (> 0, > 0) when a == b:
            Console.WriteLine($"Both measurements are valid and equal to {a}.");
            break;

        case (> 0, > 0):
            Console.WriteLine($"First measurement is {a}, second measurement is {b}.");
            break;

        default:
            Console.WriteLine("One or both measurements are not valid.");
            break;
    }
}
```

The preceding example uses [positional patterns](#) with nested [relational patterns](#).

Language version support

The `switch` statement supports pattern matching beginning with C# 7.0.

In C# 6 and earlier, you use the `switch` statement with the following limitations:

- A match expression must be of one of the following types: [char](#), [string](#), [bool](#), an [integral numeric](#) type, or an [enum](#) type.
- Only constant expressions are allowed in `case` labels.

C# language specification

For more information, see the following sections of the [C# language specification](#):

- [The `if` statement](#)
- [The `switch` statement](#)

For more information about features introduced in C# 7.0 and later, see the following feature proposal notes:

- [Switch statement \(Pattern matching for C# 7.0\)](#)

See also

- [C# reference](#)
- [Conditional operator `?:`](#)
- [Logical operators](#)
- [Patterns](#)
- [switch expression](#)

Jump statements (C# reference)

12/28/2021 • 5 minutes to read • [Edit Online](#)

The following statements unconditionally transfer control:

- The `break` statement: terminates the closest enclosing [iteration statement](#) or `switch` statement.
- The `continue` statement: starts a new iteration of the closest enclosing [iteration statement](#).
- The `return` statement: terminates execution of the function in which it appears and returns control to the caller.
- The `goto` statement: transfers control to a statement that is marked by a label.

For information about the `throw` statement that throws an exception and unconditionally transfers control as well, see [throw](#).

The `break` statement

The `break` statement terminates the closest enclosing [iteration statement](#) (that is, `for`, `foreach`, `while`, or `do` loop) or `switch` statement. The `break` statement transfers control to the statement that follows the terminated statement, if any.

```
int[] numbers = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
foreach (int number in numbers)
{
    if (number == 3)
    {
        break;
    }

    Console.Write($"{number} ");
}
Console.WriteLine();
Console.WriteLine("End of the example.");
// Output:
// 0 1 2
// End of the example.
```

In nested loops, the `break` statement terminates only the innermost loop that contains it, as the following example shows:

```

for (int outer = 0; outer < 5; outer++)
{
    for (int inner = 0; inner < 5; inner++)
    {
        if (inner > outer)
        {
            break;
        }

        Console.Write($"{inner} ");
    }
    Console.WriteLine();
}
// Output:
// 0
// 0 1
// 0 1 2
// 0 1 2 3
// 0 1 2 3 4

```

When you use the `switch` statement inside a loop, a `break` statement at the end of a switch section transfers control only out of the `switch` statement. The loop that contains the `switch` statement is unaffected, as the following example shows:

```

double[] measurements = { -4, 5, 30, double.NaN };
foreach (double measurement in measurements)
{
    switch (measurement)
    {
        case < 0.0:
            Console.WriteLine($"Measured value is {measurement}; too low.");
            break;

        case > 15.0:
            Console.WriteLine($"Measured value is {measurement}; too high.");
            break;

        case double.NaN:
            Console.WriteLine("Failed measurement.");
            break;

        default:
            Console.WriteLine($"Measured value is {measurement}.");
            break;
    }
}
// Output:
// Measured value is -4; too low.
// Measured value is 5.
// Measured value is 30; too high.
// Failed measurement.

```

The `continue` statement

The `continue` statement starts a new iteration of the closest enclosing [iteration statement](#) (that is, `for`, `foreach`, `while`, or `do` loop), as the following example shows:

```

for (int i = 0; i < 5; i++)
{
    Console.WriteLine($"Iteration {i}: ");

    if (i < 3)
    {
        Console.WriteLine("skip");
        continue;
    }

    Console.WriteLine("done");
}
// Output:
// Iteration 0: skip
// Iteration 1: skip
// Iteration 2: skip
// Iteration 3: done
// Iteration 4: done

```

The `return` statement

The `return` statement terminates execution of the function in which it appears and returns control and the function's result, if any, to the caller.

If a function member doesn't compute a value, you use the `return` statement without expression, as the following example shows:

```

Console.WriteLine("First call:");
DisplayIfNecessary(6);

Console.WriteLine("Second call:");
DisplayIfNecessary(5);

void DisplayIfNecessary(int number)
{
    if (number % 2 == 0)
    {
        return;
    }

    Console.WriteLine(number);
}
// Output:
// First call:
// Second call:
// 5

```

As the preceding example shows, you typically use the `return` statement without expression to terminate a function member early. If a function member doesn't contain the `return` statement, it terminates after its last statement is executed.

If a function member computes a value, you use the `return` statement with an expression, as the following example shows:

```
double surfaceArea = CalculateCylinderSurfaceArea(1, 1);
Console.WriteLine($"{surfaceArea:F2}"); // output: 12.57

double CalculateCylinderSurfaceArea(double baseRadius, double height)
{
    double baseArea = Math.PI * baseRadius * baseRadius;
    double sideArea = 2 * Math.PI * baseRadius * height;
    return 2 * baseArea + sideArea;
}
```

When the `return` statement has an expression, that expression must be implicitly convertible to the return type of a function member unless it's `async`. In the case of an `async` function, the expression must be implicitly convertible to the type argument of `Task<TResult>` or `ValueTask<TResult>`, whichever is the return type of the function. If the return type of an `async` function is `Task` or `ValueTask`, you use the `return` statement without expression.

By default, the `return` statement returns the value of an expression. Beginning with C# 7.0, you can return a reference to a variable. To do that, use the `return` statement with the `ref` keyword, as the following example shows:

```
var xs = new int[] { 10, 20, 30, 40 };
ref int found = ref FindFirst(xs, s => s == 30);
found = 0;
Console.WriteLine(string.Join(" ", xs)); // output: 10 20 0 40

ref int FindFirst(int[] numbers, Func<int, bool> predicate)
{
    for (int i = 0; i < numbers.Length; i++)
    {
        if (predicate(numbers[i]))
        {
            return ref numbers[i];
        }
    }
    throw new InvalidOperationException("No element satisfies the given condition.");
}
```

For more information about ref returns, see [Ref returns and ref locals](#).

The `goto` statement

The `goto` statement transfers control to a statement that is marked by a label, as the following example shows:


```

var matrices = new Dictionary<string, int[][]>
{
    ["A"] = new[]
    {
        new[] { 1, 2, 3, 4 },
        new[] { 4, 3, 2, 1 }
    },
    ["B"] = new[]
    {
        new[] { 5, 6, 7, 8 },
        new[] { 8, 7, 6, 5 }
    },
};

CheckMatrices(matrices, 4);

void CheckMatrices(Dictionary<string, int[][]> matrixLookup, int target)
{
    foreach (var (key, matrix) in matrixLookup)
    {
        for (int row = 0; row < matrix.Length; row++)
        {
            for (int col = 0; col < matrix[row].Length; col++)
            {
                if (matrix[row][col] == target)
                {
                    goto Found;
                }
            }
        }
        Console.WriteLine($"Not found {target} in matrix {key}.");
        continue;

    Found:
        Console.WriteLine($"Found {target} in matrix {key}.");
    }
}

// Output:
// Found 4 in matrix A.
// Not found 4 in matrix B.

```

As the preceding example shows, you can use the `goto` statement to get out of a nested loop.

TIP

When you work with nested loops, consider refactoring separate loops into separate methods. That may lead to a simpler, more readable code without the `goto` statement.

You can also use the `goto` statement in the `switch statement` to transfer control to a switch section with a constant case label, as the following example shows:

```

using System;

public enum CoffeChoice
{
    Plain,
    WithMilk,
    WithIceCream,
}

public class GotoInSwitchExample
{
    public static void Main()
    {
        Console.WriteLine(CalculatePrice(CoffeChoice.Plain)); // output: 10.0
        Console.WriteLine(CalculatePrice(CoffeChoice.WithMilk)); // output: 15.0
        Console.WriteLine(CalculatePrice(CoffeChoice.WithIceCream)); // output: 17.0
    }

    private static decimal CalculatePrice(CoffeChoice choice)
    {
        decimal price = 0;
        switch (choice)
        {
            case CoffeChoice.Plain:
                price += 10.0m;
                break;

            case CoffeChoice.WithMilk:
                price += 5.0m;
                goto case CoffeChoice.Plain;

            case CoffeChoice.WithIceCream:
                price += 7.0m;
                goto case CoffeChoice.Plain;
        }
        return price;
    }
}

```

Within the `switch` statement, you can also use the statement `goto default;` to transfer control to the switch section with the `default` label.

If a label with the given name doesn't exist in the current function member, or if the `goto` statement is not within the scope of the label, a compile-time error occurs. That is, you can't use the `goto` statement to transfer control out of the current function member or into any nested scope, for example, a `try` block.

C# language specification

For more information, see the following sections of the [C# language specification](#):

- [The `break` statement](#)
- [The `continue` statement](#)
- [The `return` statement](#)
- [The `goto` statement](#)

See also

- [C# reference](#)
- [yield statement](#)

lock statement (C# reference)

12/28/2021 • 2 minutes to read • [Edit Online](#)

The `lock` statement acquires the mutual-exclusion lock for a given object, executes a statement block, and then releases the lock. While a lock is held, the thread that holds the lock can again acquire and release the lock. Any other thread is blocked from acquiring the lock and waits until the lock is released.

The `lock` statement is of the form

```
lock (x)
{
    // Your code...
}
```

where `x` is an expression of a [reference type](#). It's precisely equivalent to

```
object __lockObj = x;
bool __lockWasTaken = false;
try
{
    System.Threading.Monitor.Enter(__lockObj, ref __lockWasTaken);
    // Your code...
}
finally
{
    if (__lockWasTaken) System.Threading.Monitor.Exit(__lockObj);
}
```

Since the code uses a [try...finally](#) block, the lock is released even if an exception is thrown within the body of a `lock` statement.

You can't use the [await operator](#) in the body of a `lock` statement.

Guidelines

When you synchronize thread access to a shared resource, lock on a dedicated object instance (for example, `private readonly object balanceLock = new object();`) or another instance that is unlikely to be used as a lock object by unrelated parts of the code. Avoid using the same lock object instance for different shared resources, as it might result in deadlock or lock contention. In particular, avoid using the following as lock objects:

- `this`, as it might be used by the callers as a lock.
- [Type](#) instances, as those might be obtained by the [typeof](#) operator or reflection.
- string instances, including string literals, as those might be [interned](#).

Hold a lock for as short time as possible to reduce lock contention.

Example

The following example defines an `Account` class that synchronizes access to its private `balance` field by locking on a dedicated `balanceLock` instance. Using the same instance for locking ensures that the `balance` field cannot be updated simultaneously by two threads attempting to call the `Debit` or `Credit` methods simultaneously.

```

using System;
using System.Threading.Tasks;

public class Account
{
    private readonly object balanceLock = new object();
    private decimal balance;

    public Account(decimal initialBalance) => balance = initialBalance;

    public decimal Debit(decimal amount)
    {
        if (amount < 0)
        {
            throw new ArgumentOutOfRangeException(nameof(amount), "The debit amount cannot be negative.");
        }

        decimal appliedAmount = 0;
        lock (balanceLock)
        {
            if (balance >= amount)
            {
                balance -= amount;
                appliedAmount = amount;
            }
        }
        return appliedAmount;
    }

    public void Credit(decimal amount)
    {
        if (amount < 0)
        {
            throw new ArgumentOutOfRangeException(nameof(amount), "The credit amount cannot be negative.");
        }

        lock (balanceLock)
        {
            balance += amount;
        }
    }

    public decimal GetBalance()
    {
        lock (balanceLock)
        {
            return balance;
        }
    }
}

class AccountTest
{
    static async Task Main()
    {
        var account = new Account(1000);
        var tasks = new Task[100];
        for (int i = 0; i < tasks.Length; i++)
        {
            tasks[i] = Task.Run(() => Update(account));
        }
        await Task.WhenAll(tasks);
        Console.WriteLine($"Account's balance is {account.GetBalance()}");
        // Output:
        // Account's balance is 2000
    }

    static void Update(Account account)
    {

```

```
decimal[] amounts = { 0, 2, -3, 6, -2, -1, 8, -5, 11, -6 };
foreach (var amount in amounts)
{
    if (amount >= 0)
    {
        account.Credit(amount);
    }
    else
    {
        account.Debit(Math.Abs(amount));
    }
}
}
```

C# language specification

For more information, see [The lock statement](#) section of the [C# language specification](#).

See also

- [C# reference](#)
- [System.Threading.Monitor](#)
- [System.Threading.SpinLock](#)
- [System.Threading.Interlocked](#)
- [Overview of synchronization primitives](#)

C# Special Characters

12/28/2021 • 2 minutes to read • [Edit Online](#)

Special characters are predefined, contextual characters that modify the program element (a literal string, an identifier, or an attribute name) to which they are prepended. C# supports the following special characters:

- `@`, the verbatim identifier character.
- `$`, the interpolated string character.

This section only includes those tokens that are not operators. See the [operators](#) section for all operators.

See also

- [C# Reference](#)
- [C# Programming Guide](#)

\$ - string interpolation (C# reference)

12/28/2021 • 5 minutes to read • [Edit Online](#)

The `$` special character identifies a string literal as an *interpolated string*. An interpolated string is a string literal that might contain *interpolation expressions*. When an interpolated string is resolved to a result string, items with interpolation expressions are replaced by the string representations of the expression results. This feature is available starting with C# 6.

String interpolation provides a more readable, convenient syntax to format strings. It's easier to read than [string composite formatting](#). Compare the following example that uses both features to produce the same output:

```
string name = "Mark";
var date = DateTime.Now;

// Composite formatting:
Console.WriteLine("Hello, {0}! Today is {1}, it's {2:HH:mm} now.", name, date.DayOfWeek, date);
// String interpolation:
Console.WriteLine($"Hello, {name}! Today is {date.DayOfWeek}, it's {date:HH:mm} now.");
// Both calls produce the same output that is similar to:
// Hello, Mark! Today is Wednesday, it's 19:40 now.
```

Structure of an interpolated string

To identify a string literal as an interpolated string, prepend it with the `$` symbol. You can't have any white space between the `$` and the `"` that starts a string literal.

The structure of an item with an interpolation expression is as follows:

```
{<interpolationExpression>[,<alignment>][:<formatString>]}
```

Elements in square brackets are optional. The following table describes each element:

ELEMENT	DESCRIPTION
<code>interpolationExpression</code>	The expression that produces a result to be formatted. String representation of <code>null</code> is String.Empty .
<code>alignment</code>	The constant expression whose value defines the minimum number of characters in the string representation of the expression result. If positive, the string representation is right-aligned; if negative, it's left-aligned. For more information, see Alignment Component .
<code>formatString</code>	A format string that is supported by the type of the expression result. For more information, see Format String Component .

The following example uses optional formatting components described above:

```

Console.WriteLine($"|{"Left",-7}|{"Right",7}|");

const int FieldWidthRightAligned = 20;
Console.WriteLine($"{Math.PI,FieldWidthRightAligned} - default formatting of the pi number");
Console.WriteLine($"{Math.PI,FieldWidthRightAligned:F3} - display only three decimal digits of the pi number");
// Expected output is:
// |Left  | Right|
//      3.14159265358979 - default formatting of the pi number
//                      3.142 - display only three decimal digits of the pi number

```

Beginning with C# 10, you can use string interpolation to initialize a constant string. All expressions used for placeholders must be constant strings. In other words, every *interpolation expression* must be a string, and it must be a compile time constant.

Special characters

To include a brace, "{" or "}", in the text produced by an interpolated string, use two braces, "{{" or "}}". For more information, see [Escaping Braces](#).

As the colon (":") has special meaning in an interpolation expression item, to use a [conditional operator](#) in an interpolation expression, enclose that expression in parentheses.

The following example shows how to include a brace in a result string. It also shows how to use a conditional operator:

```

string name = "Horace";
int age = 34;
Console.WriteLine($"He asked, \"Is your name {name}?\", but didn't wait for a reply :-{{}}");
Console.WriteLine($"{name} is {age} year{(age == 1 ? "" : "s")} old.");
// Expected output is:
// He asked, "Is your name Horace?", but didn't wait for a reply :-{
// Horace is 34 years old.

```

An interpolated verbatim string starts with the `$` character followed by the `@` character. For more information about verbatim strings, see the [string](#) and [verbatim identifier](#) articles.

NOTE

Starting with C# 8.0, you can use the `$` and `@` tokens in any order: both `@$"..."` and `@$"..."` are valid interpolated verbatim strings. In earlier C# versions, the `$` token must appear before the `@` token.

Implicit conversions and how to specify `IFormatProvider` implementation

There are three implicit conversions from an interpolated string:

1. Conversion of an interpolated string to a [String](#) instance. The string is the result of interpolated string resolution. All interpolation expression items are replaced with the properly formatted string representations of their results. This conversion uses the [CurrentCulture](#) to format expression results.
2. Conversion of an interpolated string to a [FormattableString](#) instance that represents a composite format string along with the expression results to be formatted. That allows you to create multiple result strings with culture-specific content from a single [FormattableString](#) instance. To do that, call one of the following methods:

- A [ToString\(\)](#) overload that produces a result string for the [CurrentCulture](#).
- An [Invariant](#) method that produces a result string for the [InvariantCulture](#).
- A [ToString\(IFormatProvider\)](#) method that produces a result string for a specified culture.

The [ToString\(IFormatProvider\)](#) provides a user-defined implementation of the [IFormatProvider](#) interface that supports custom formatting. For more information, see the [Custom formatting with ICustomFormatter](#) section of the [Formatting types in .NET](#) article.

3. Conversion of an interpolated string to an [IFormattable](#) instance that also allows you to create multiple result strings with culture-specific content from a single [IFormattable](#) instance.

The following example uses implicit conversion to [FormattableString](#) to create culture-specific result strings:

```
double speedOfLight = 299792.458;
FormattableString message = $"The speed of light is {speedOfLight:N3} km/s.";

System.Globalization.CultureInfo.CurrentCulture = System.Globalization.CultureInfo.GetCultureInfo("nl-NL");
string messageInCurrentCulture = message.ToString();

var specificCulture = System.Globalization.CultureInfo.GetCultureInfo("en-IN");
string messageInSpecificCulture = message.ToString(specificCulture);

string messageInInvariantCulture = FormattableString.Invariant(message);

Console.WriteLine($"{System.Globalization.CultureInfo.CurrentCulture,-10} {messageInCurrentCulture}");
Console.WriteLine($"{specificCulture,-10} {messageInSpecificCulture}");
Console.WriteLine($"{Invariant,-10} {messageInInvariantCulture}");
// Expected output is:
// nl-NL      The speed of light is 299.792,458 km/s.
// en-IN      The speed of light is 2,99,792.458 km/s.
// Invariant  The speed of light is 299,792.458 km/s.
```

Additional resources

If you're new to string interpolation, see the [String interpolation in C#](#) interactive tutorial. You can also check another [String interpolation in C#](#) tutorial. That tutorial demonstrates how to use interpolated strings to produce formatted strings.

Compilation of interpolated strings

If an interpolated string has the type `string`, it's typically transformed into a [String.Format](#) method call. The compiler may replace [String.Format](#) with [String.Concat](#) if the analyzed behavior would be equivalent to concatenation.

If an interpolated string has the type [IFormattable](#) or [FormattableString](#), the compiler generates a call to the [FormattableStringFactory.Create](#) method.

Beginning with C# 10, when an interpolated string is used, the compiler checks if the interpolated string is assigned to a type that satisfies the *interpolated string handler pattern*. An *interpolated string handler* is a custom type that converts the interpolated string into a string. An interpolated string handler is an advanced scenario, typically used for performance reasons. You can learn about the requirements to build an interpolated string handler in the language specification for [interpolated string improvements](#). You can build one following the [interpolated string handler tutorial](#) in the What's new in C# section. In .NET 6, when you use an interpolated string for an argument of type `string`, the interpolated string is processed by the [System.Runtime.CompilerServices.DefaultInterpolatedStringHandler](#).

NOTE

One side effect of interpolated string handlers is that a custom handler, including [System.Runtime.CompilerServices.DefaultInterpolatedStringHandler](#), may not evaluate all the expressions used as placeholders in the interpolated string under all conditions. That means side-effects in those expressions may not occur.

C# language specification

For more information, see the [Interpolated strings](#) section of the [C# language specification](#).

See also

- [C# reference](#)
- [C# special characters](#)
- [Strings](#)
- [Standard numeric format strings](#)
- [Composite formatting](#)
- [String.Format](#)

@ (C# Reference)

12/28/2021 • 2 minutes to read • [Edit Online](#)

The `@` special character serves as a verbatim identifier. It can be used in the following ways:

1. To enable C# keywords to be used as identifiers. The `@` character prefixes a code element that the compiler is to interpret as an identifier rather than a C# keyword. The following example uses the `@` character to define an identifier named `for` that it uses in a `for` loop.

```
string[] @for = { "John", "James", "Joan", "Jamie" };
for (int ctr = 0; ctr < @for.Length; ctr++)
{
    Console.WriteLine($"Here is your gift, {@for[ctr]}!");
}
// The example displays the following output:
//      Here is your gift, John!
//      Here is your gift, James!
//      Here is your gift, Joan!
//      Here is your gift, Jamie!
```

2. To indicate that a string literal is to be interpreted verbatim. The `@` character in this instance defines a *verbatim string literal*. Simple escape sequences (such as `"\"` for a backslash), hexadecimal escape sequences (such as `"\x0041"` for an uppercase A), and Unicode escape sequences (such as `"\u0041"` for an uppercase A) are interpreted literally. Only a quote escape sequence (`"\"`) is not interpreted literally; it produces one double quotation mark. Additionally, in case of a verbatim *interpolated string* brace escape sequences (`{` and `}`) are not interpreted literally; they produce single brace characters. The following example defines two identical file paths, one by using a regular string literal and the other by using a verbatim string literal. This is one of the more common uses of verbatim string literals.

```
string filename1 = @"c:\documents\files\u0066.txt";
string filename2 = "c:\\documents\\files\\u0066.txt";

Console.WriteLine(filename1);
Console.WriteLine(filename2);
// The example displays the following output:
//      c:\documents\files\u0066.txt
//      c:\documents\files\u0066.txt
```

The following example illustrates the effect of defining a regular string literal and a verbatim string literal that contain identical character sequences.

```
string s1 = "He said, \"This is the last \u0063hance\u0021\"";
string s2 = @"He said, ""This is the last \u0063hance\u0021""";

Console.WriteLine(s1);
Console.WriteLine(s2);
// The example displays the following output:
//      He said, "This is the last chance!"
//      He said, "This is the last \u0063hance\u0021"
```

3. To enable the compiler to distinguish between attributes in cases of a naming conflict. An attribute is a class that derives from `Attribute`. Its type name typically includes the suffix `Attribute`, although the compiler does not enforce this convention. The attribute can then be referenced in code either by its full

type name (for example, `[InfoAttribute]` or its shortened name (for example, `[Info]`). However, a naming conflict occurs if two shortened attribute type names are identical, and one type name includes the **Attribute** suffix but the other does not. For example, the following code fails to compile because the compiler cannot determine whether the `Info` or `InfoAttribute` attribute is applied to the `Example` class. See [CS1614](#) for more information.

```
using System;

[AttributeUsage(AttributeTargets.Class)]
public class Info : Attribute
{
    private string information;

    public Info(string info)
    {
        information = info;
    }
}

[AttributeUsage(AttributeTargets.Method)]
public class InfoAttribute : Attribute
{
    private string information;

    public InfoAttribute(string info)
    {
        information = info;
    }
}

[Info("A simple executable.")] // Generates compiler error CS1614. Ambiguous Info and InfoAttribute.
// Prepend '@' to select 'Info' ([@Info("A simple executable.")] ). Specify the full name
'InfoAttribute' to select it.
public class Example
{
    [InfoAttribute("The entry point.")]
    public static void Main()
    {
    }
}
```

See also

- [C# Reference](#)
- [C# Programming Guide](#)
- [C# Special Characters](#)

Assembly level attributes interpreted by the C# compiler

12/28/2021 • 2 minutes to read • [Edit Online](#)

Most attributes are applied to specific language elements such as classes or methods; however, some attributes are global—they apply to an entire assembly or module. For example, the [AssemblyVersionAttribute](#) attribute can be used to embed version information into an assembly, like this:

```
[assembly: AssemblyVersion("1.0.0.0")]
```

Global attributes appear in the source code after any top level `using` directives and before any type, module, or namespace declarations. Global attributes can appear in multiple source files, but the files must be compiled in a single compilation pass. Visual Studio adds global attributes to the AssemblyInfo.cs file in .NET Framework projects. These attributes aren't added to .NET Core projects.

Assembly attributes are values that provide information about an assembly. They fall into the following categories:

- Assembly identity attributes
- Informational attributes
- Assembly manifest attributes

Assembly identity attributes

Three attributes (with a strong name, if applicable) determine the identity of an assembly: name, version, and culture. These attributes form the full name of the assembly and are required when you reference it in code. You can set an assembly's version and culture using attributes. However, the name value is set by the compiler, the Visual Studio IDE in the [Assembly Information Dialog Box](#), or the Assembly Linker (AL.exe) when the assembly is created. The assembly name is based on the assembly manifest. The [AssemblyFlagsAttribute](#) attribute specifies whether multiple copies of the assembly can coexist.

The following table shows the identity attributes.

ATTRIBUTE	PURPOSE
AssemblyVersionAttribute	Specifies the version of an assembly.
AssemblyCultureAttribute	Specifies which culture the assembly supports.
AssemblyFlagsAttribute	Specifies whether an assembly supports side-by-side execution on the same computer, in the same process, or in the same application domain.

Informational attributes

You use informational attributes to provide additional company or product information for an assembly. The following table shows the informational attributes defined in the [System.Reflection](#) namespace.

ATTRIBUTE	PURPOSE
AssemblyProductAttribute	Specifies a product name for an assembly manifest.
AssemblyTrademarkAttribute	Specifies a trademark for an assembly manifest.
AssemblyInformationalVersionAttribute	Specifies an informational version for an assembly manifest.
AssemblyCompanyAttribute	Specifies a company name for an assembly manifest.
AssemblyCopyrightAttribute	Defines a custom attribute that specifies a copyright for an assembly manifest.
AssemblyFileVersionAttribute	Sets a specific version number for the Win32 file version resource.
CLSCompliantAttribute	Indicates whether the assembly is compliant with the Common Language Specification (CLS).

Assembly manifest attributes

You can use assembly manifest attributes to provide information in the assembly manifest. The attributes include title, description, default alias, and configuration. The following table shows the assembly manifest attributes defined in the [System.Reflection](#) namespace.

ATTRIBUTE	PURPOSE
AssemblyTitleAttribute	Specifies an assembly title for an assembly manifest.
AssemblyDescriptionAttribute	Specifies an assembly description for an assembly manifest.
AssemblyConfigurationAttribute	Specifies an assembly configuration (such as retail or debug) for an assembly manifest.
AssemblyDefaultAliasAttribute	Defines a friendly default alias for an assembly manifest

Determine caller information using attributes interpreted by the C# compiler

12/28/2021 • 4 minutes to read • [Edit Online](#)

Using info attributes, you obtain information about the caller to a method. You obtain the file path of the source code, the line number in the source code, and the member name of the caller. To obtain member caller information, you use attributes that are applied to optional parameters. Each optional parameter specifies a default value. The following table lists the Caller Info attributes that are defined in the [System.Runtime.CompilerServices](#) namespace:

ATTRIBUTE	DESCRIPTION	TYPE
CallerFilePathAttribute	Full path of the source file that contains the caller. The full path is the path at compile time.	<code>String</code>
CallerLineNumberAttribute	Line number in the source file from which the method is called.	<code>Integer</code>
CallerMemberNameAttribute	Method name or property name of the caller.	<code>String</code>
CallerArgumentExpressionAttribute	String representation of the argument expression.	<code>String</code>

This information helps you with tracing and debugging, and helps you to create diagnostic tools. The following example shows how to use caller info attributes. On each call to the `TraceMessage` method, the caller information is inserted for the arguments to the optional parameters.

```
public void DoProcessing()
{
    TraceMessage("Something happened.");
}

public void TraceMessage(string message,
    [CallerMemberName] string memberName = "",
    [CallerFilePath] string sourceFilePath = "",
    [CallerLineNumber] int sourceLineNumber = 0)
{
    Trace.WriteLine("message: " + message);
    Trace.WriteLine("member name: " + memberName);
    Trace.WriteLine("source file path: " + sourceFilePath);
    Trace.WriteLine("source line number: " + sourceLineNumber);
}

// Sample Output:
// message: Something happened.
// member name: DoProcessing
// source file path: c:\Visual Studio Projects\CallerInfoCS\CallerInfoCS\Form1.cs
// source line number: 31
```

You specify an explicit default value for each optional parameter. You can't apply caller info attributes to parameters that aren't specified as optional. The caller info attributes don't make a parameter optional. Instead,

they affect the default value that's passed in when the argument is omitted. Caller info values are emitted as literals into the Intermediate Language (IL) at compile time. Unlike the results of the [StackTrace](#) property for exceptions, the results aren't affected by obfuscation. You can explicitly supply the optional arguments to control the caller information or to hide caller information.

Member names

You can use the `CallerMemberName` attribute to avoid specifying the member name as a `String` argument to the called method. By using this technique, you avoid the problem that **Rename Refactoring** doesn't change the `String` values. This benefit is especially useful for the following tasks:

- Using tracing and diagnostic routines.
- Implementing the [INotifyPropertyChanged](#) interface when binding data. This interface allows the property of an object to notify a bound control that the property has changed. The control can display the updated information. Without the `CallerMemberName` attribute, you must specify the property name as a literal.

The following chart shows the member names that are returned when you use the `CallerMemberName` attribute.

CALLS OCCUR WITHIN	MEMBER NAME RESULT
Method, property, or event	The name of the method, property, or event from which the call originated.
Constructor	The string ".ctor"
Static constructor	The string ".cctor"
Finalizer	The string "Finalize"
User-defined operators or conversions	The generated name for the member, for example, "op_Addition".
Attribute constructor	The name of the method or property to which the attribute is applied. If the attribute is any element within a member (such as a parameter, a return value, or a generic type parameter), this result is the name of the member that's associated with that element.
No containing member (for example, assembly-level or attributes that are applied to types)	The default value of the optional parameter.

Argument expressions

You use the [System.Runtime.CompilerServices.CallerArgumentExpressionAttribute](#) when you want the expression passed as an argument. Diagnostic libraries may want to provide more details about the *expressions* passed to arguments. By providing the expression that triggered the diagnostic, in addition to the parameter name, developers have more details about the condition that triggered the diagnostic. That extra information makes it easier to fix.

The following example shows how you can provide detailed information about the argument when it's invalid:


```
public static void ValidateArgument(string parameterName, bool condition,
[CallerArgumentExpression("condition")] string? message=null)
{
    if (!condition)
    {
        throw new ArgumentException($"Argument failed validation: <{message}>", parameterName);
    }
}
```

You would invoke it as shown in the following example:

```
public void Operation(Action func)
{
    Utilities.ValidateArgument(nameof(func), func is not null);
    func();
}
```

The expression used for `condition` is injected by the compiler into the `message` argument. When a developer calls `Operation` with a `null` argument, the following message is stored in the `ArgumentException`:

```
Argument failed validation: <func is not null>
```

This attribute enables you to write diagnostic utilities that provide more details. Developers can more quickly understand what changes are needed. You can also use the [CallerArgumentExpressionAttribute](#) to determine what expression was used as the receiver for extension methods. The following method samples a sequence at regular intervals. If the sequence has fewer elements than the frequency, it reports an error:

```
public static IEnumerable<T> Sample<T>(this IEnumerable<T> sequence, int frequency,
[CallerArgumentExpression("sequence")] string? message = null)
{
    if (sequence.Count() < frequency)
        throw new ArgumentException($"Expression doesn't have enough elements: {message}",
nameof(sequence));
    int i = 0;
    foreach (T item in sequence)
    {
        if (i++ % frequency == 0)
            yield return item;
    }
}
```

You could call this method as follows:

```
sample = Enumerable.Range(0, 10).Sample(100);
```

The preceding example would throw an [ArgumentException](#) whose message is the following text:

```
Expression doesn't have enough elements: Enumerable.Range(0, 10) (Parameter 'sequence')
```

See also

- [Named and Optional Arguments](#)
- [System.Reflection](#)
- [Attribute](#)

- [Attributes](#)

Attributes for null-state static analysis interpreted by the C# compiler

12/28/2021 • 14 minutes to read • [Edit Online](#)

In a nullable enabled context, the compiler performs static analysis of code to determine the *null-state* of all reference type variables:

- *not-null*: Static analysis determines that a variable has a non-null value.
- *maybe-null*: Static analysis can't determine that a variable is assigned a non-null value.

These states enable the compiler to provide warnings when you may dereference a null value, throwing a [System.NullReferenceException](#). These attributes provide the compiler with semantic information about the *null-state* of arguments, return values, and object members based on the state of arguments and return values. The compiler provides more accurate warnings when your APIs have been properly annotated with this semantic information.

This article provides a brief description of each of the nullable reference type attributes and how to use them.

Let's start with an example. Imagine your library has the following API to retrieve a resource string. This method was originally written before C# 8.0 and nullable annotations:

```
bool TryGetMessage(string key, out string message)
{
    if (_messageMap.ContainsKey(key))
        message = _messageMap[key];
    else
        message = null;
    return message != null;
}
```

The preceding example follows the familiar `Try*` pattern in .NET. There are two reference parameters for this API: the `key` and the `message`. This API has the following rules relating to the *null-state* of these parameters:

- Callers shouldn't pass `null` as the argument for `key`.
- Callers can pass a variable whose value is `null` as the argument for `message`.
- If the `TryGetMessage` method returns `true`, the value of `message` isn't null. If the return value is `false`, the value of `message` is null.

The rule for `key` can be expressed succinctly in C# 8.0: `key` should be a non-nullable reference type. The `message` parameter is more complex. It allows a variable that is `null` as the argument, but guarantees, on success, that the `out` argument isn't `null`. For these scenarios, you need a richer vocabulary to describe the expectations. The `NotNullWhen` attribute, described below describes the *null-state* for the argument used for the `message` parameter.

NOTE

Adding these attributes gives the compiler more information about the rules for your API. When calling code is compiled in a nullable enabled context, the compiler will warn callers when they violate those rules. These attributes don't enable more checks on your implementation.

ATTRIBUTE	CATEGORY	MEANING
<code>AllowNull</code>	Precondition	A non-nullable parameter, field, or property may be null.
<code>DisallowNull</code>	Precondition	A nullable parameter, field, or property should never be null.
<code>MaybeNull</code>	Postcondition	A non-nullable parameter, field, property, or return value may be null.
<code>NotNull</code>	Postcondition	A nullable parameter, field, property, or return value will never be null.
<code>MaybeNullWhen</code>	Conditional postcondition	A non-nullable argument may be null when the method returns the specified <code>bool</code> value.
<code>NotNullWhen</code>	Conditional postcondition	A nullable argument won't be null when the method returns the specified <code>bool</code> value.
<code>NotNullIfNotNull</code>	Conditional postcondition	A return value, property, or argument isn't null if the argument for the specified parameter isn't null.
<code>MemberNotNull</code>	Method and property helper methods	The listed member won't be null when the method returns.
<code>MemberNotNullWhen</code>	Method and property helper methods	The listed member won't be null when the method returns the specified <code>bool</code> value.
<code>DoesNotReturn</code>	Unreachable code	A method or property never returns. In other words, it always throws an exception.
<code>DoesNotReturnIf</code>	Unreachable code	This method or property never returns if the associated <code>bool</code> parameter has the specified value.

The preceding descriptions are a quick reference to what each attribute does. The following sections describe the behavior and meaning of these attributes more thoroughly.

Preconditions: `AllowNull` and `DisallowNull`

Consider a read/write property that never returns `null` because it has a reasonable default value. Callers pass `null` to the set accessor when setting it to that default value. For example, consider a messaging system that asks for a screen name in a chat room. If none is provided, the system generates a random name:

```
public string ScreenName
{
    get => _screenName;
    set => _screenName = value ?? GenerateRandomScreenName();
}
private string _screenName;
```

When you compile the preceding code in a nullable oblivious context, everything is fine. Once you enable nullable reference types, the `ScreenName` property becomes a non-nullable reference. That's correct for the `get` accessor: it never returns `null`. Callers don't need to check the returned property for `null`. But now setting the property to `null` generates a warning. To support this type of code, you add the [System.Diagnostics.CodeAnalysis.AllowNullAttribute](#) to the property, as shown in the following code:

```
[AllowNull]
public string ScreenName
{
    get => _screenName;
    set => _screenName = value ?? GenerateRandomScreenName();
}
private string _screenName = GenerateRandomScreenName();
```

You may need to add a `using` directive for [System.Diagnostics.CodeAnalysis](#) to use this and other attributes discussed in this article. The attribute is applied to the property, not the `set` accessor. The `AllowNull` attribute specifies *pre-conditions*, and only applies to arguments. The `get` accessor has a return value, but no parameters. Therefore, the `AllowNull` attribute only applies to the `set` accessor.

The preceding example demonstrates what to look for when adding the `AllowNull` attribute on an argument:

1. The general contract for that variable is that it shouldn't be `null`, so you want a non-nullable reference type.
2. There are scenarios for a caller to pass `null` as the argument, though they aren't the most common usage.

Most often you'll need this attribute for properties, or `in`, `out`, and `ref` arguments. The `AllowNull` attribute is the best choice when a variable is typically non-null, but you need to allow `null` as a precondition.

Contrast that with scenarios for using `DisallowNull`: You use this attribute to specify that an argument of a nullable reference type shouldn't be `null`. Consider a property where `null` is the default value, but clients can only set it to a non-null value. Consider the following code:

```
public string ReviewComment
{
    get => _comment;
    set => _comment = value ?? throw new ArgumentNullException(nameof(value), "Cannot set to null");
}
string _comment;
```

The preceding code is the best way to express your design that the `ReviewComment` could be `null`, but can't be set to `null`. Once this code is nullable aware, you can express this concept more clearly to callers using the [System.Diagnostics.CodeAnalysis.DisallowNullAttribute](#):

```
[DisallowNull]
public string? ReviewComment
{
    get => _comment;
    set => _comment = value ?? throw new ArgumentNullException(nameof(value), "Cannot set to null");
}
string? _comment;
```

In a nullable context, the `ReviewComment` `get` accessor could return the default value of `null`. The compiler warns that it must be checked before access. Furthermore, it warns callers that, even though it could be `null`, callers shouldn't explicitly set it to `null`. The `DisallowNull` attribute also specifies a *pre-condition*, it doesn't affect the `get` accessor. You use the `DisallowNull` attribute when you observe these characteristics about:

1. The variable could be `null` in core scenarios, often when first instantiated.
2. The variable shouldn't be explicitly set to `null`.

These situations are common in code that was originally *null oblivious*. It may be that object properties are set in two distinct initialization operations. It may be that some properties are set only after some asynchronous work has completed.

The `AllowNull` and `DisallowNull` attributes enable you to specify that preconditions on variables may not match the nullable annotations on those variables. These provide more detail about the characteristics of your API. This additional information helps callers use your API correctly. Remember you specify preconditions using the following attributes:

- [AllowNull](#): A non-nullable argument may be null.
- [DisallowNull](#): A nullable argument should never be null.

Postconditions: `MaybeNull` and `NotNull`

Suppose you have a method with the following signature:

```
public Customer FindCustomer(string lastName, string firstName)
```

You've likely written a method like this to return `null` when the name sought wasn't found. The `null` clearly indicates that the record wasn't found. In this example, you'd likely change the return type from `Customer` to `Customer?`. Declaring the return value as a nullable reference type specifies the intent of this API clearly:

```
public Customer? FindCustomer(string lastName, string firstName)
```

For reasons covered under [Generics nullability](#) that technique may not produce the static analysis that matches your API. You may have a generic method that follows a similar pattern:

```
public T Find<T>(IEnumerable<T> sequence, Func<T, bool> predicate)
```

The method returns `null` when the sought item isn't found. You can clarify that the method returns `null` when an item isn't found by adding the `MaybeNull` annotation to the method return:

```
[return: MaybeNull]
public T Find<T>(IEnumerable<T> sequence, Func<T, bool> predicate)
```

The preceding code informs callers that the return value *may* actually be null. It also informs the compiler that

the method may return a `null` expression even though the type is non-nullable. When you have a generic method that returns an instance of its type parameter, `T`, you can express that it never returns `null` by using the `NotNull` attribute.

You can also specify that a return value or an argument isn't null even though the type is a nullable reference type. The following method is a helper method that throws if its first argument is `null`:

```
public static void ThrowWhenNull(object value, string valueExpression = "")
{
    if (value is null) throw new ArgumentNullException(nameof(value), valueExpression);
}
```

You could call this routine as follows:

```
public static void LogMessage(string? message)
{
    ThrowWhenNull(message, $"{nameof(message)} must not be null");

    Console.WriteLine(message.Length);
}
```

After enabling null reference types, you want to ensure that the preceding code compiles without warnings. When the method returns, the `value` parameter is guaranteed to be not null. However, it's acceptable to call `ThrowWhenNull` with a null reference. You can make `value` a nullable reference type, and add the `NotNull` post-condition to the parameter declaration:

```
public static void ThrowWhenNull([NotNull] object? value, string valueExpression = "")
{
    _ = value ?? throw new ArgumentNullException(nameof(value), valueExpression);
    // other logic elided
}
```

The preceding code expresses the existing contract clearly: Callers can pass a variable with the `null` value, but the argument is guaranteed to never be null if the method returns without throwing an exception.

You specify unconditional postconditions using the following attributes:

- **MaybeNull**: A non-nullable return value may be null.
- **NotNull**: A nullable return value will never be null.

Conditional post-conditions: `NotNullWhen`, `MaybeNullWhen`, and `NotNullIfNotNull`

You're likely familiar with the `string` method `String.IsNullOrEmpty(String)`. This method returns `true` when the argument is null or an empty string. It's a form of null-check: Callers don't need to null-check the argument if the method returns `false`. To make a method like this nullable aware, you'd set the argument to a nullable reference type, and add the `NotNullWhen` attribute:

```
bool IsNullOrEmpty([NotNullWhen(false)] string? value)
```

That informs the compiler that any code where the return value is `false` doesn't need null checks. The addition of the attribute informs the compiler's static analysis that `IsNullOrEmpty` performs the necessary null check: when it returns `false`, the argument isn't `null`.

```
string? userInput = GetUserInput();
if (!string.IsNullOrEmpty(userInput))
{
    int messageLength = userInput.Length; // no null check needed.
}
// null check needed on userInput here.
```

The [String.IsNullOrEmpty\(String\)](#) method will be annotated as shown above for .NET Core 3.0. You may have similar methods in your codebase that check the state of objects for null values. The compiler won't recognize custom null check methods, and you'll need to add the annotations yourself. When you add the attribute, the compiler's static analysis knows when the tested variable has been null checked.

Another use for these attributes is the `Try*` pattern. The postconditions for `ref` and `out` arguments are communicated through the return value. Consider this method shown earlier (in a nullable disabled context):

```
bool TryGetMessage(string key, out string message)
{
    if (_messageMap.ContainsKey(key))
        message = _messageMap[key];
    else
        message = null;
    return message != null;
}
```

The preceding method follows a typical .NET idiom: the return value indicates if `message` was set to the found value or, if no message is found, to the default value. If the method returns `true`, the value of `message` isn't null; otherwise, the method sets `message` to null.

In a nullable enabled context, You can communicate that idiom using the `NotNullWhen` attribute. When you annotate parameters for nullable reference types, make `message` a `string?` and add an attribute:

```
bool TryGetMessage(string key, [NotNullWhen(true)] out string? message)
{
    if (_messageMap.ContainsKey(key))
        message = _messageMap[key];
    else
        message = null;
    return message is not null;
}
```

In the preceding example, the value of `message` is known to be not null when `TryGetMessage` returns `true`. You should annotate similar methods in your codebase in the same way: the arguments could equal `null`, and are known to be not null when the method returns `true`.

There's one final attribute you may also need. Sometimes the null state of a return value depends on the null state of one or more arguments. These methods will return a non-null value whenever certain arguments aren't `null`. To correctly annotate these methods, you use the `NotNullIfNotNull` attribute. Consider the following method:

```
string GetTopLevelDomainFromFullUrl(string url)
```

If the `url` argument isn't null, the output isn't `null`. Once nullable references are enabled, you need to add more annotations if your API may accept a null argument. You could annotate the return type as shown in the following code:


```
string? GetTopLevelDomainFromFullUrl(string? url)
```

That also works, but will often force callers to implement extra `null` checks. The contract is that the return value would be `null` only when the argument `url` is `null`. To express that contract, you would annotate this method as shown in the following code:

```
[return: NotNullIfNotNull("url")]
string? GetTopLevelDomainFromFullUrl(string? url)
```

The return value and the argument have both been annotated with the `?` indicating that either could be `null`. The attribute further clarifies that the return value won't be null when the `url` argument isn't `null`.

You specify conditional postconditions using these attributes:

- **MaybeNullWhen**: A non-nullable argument may be null when the method returns the specified `bool` value.
- **NotNullWhen**: A nullable argument won't be null when the method returns the specified `bool` value.
- **NotNullIfNotNull**: A return value isn't null if the argument for the specified parameter isn't null.

Helper methods: `MemberNotNull` and `MemberNotNullWhen`

These attributes specify your intent when you've refactored common code from constructors into helper methods. The C# compiler analyzes constructors and field initializers to make sure that all non-nullable reference fields have been initialized before each constructor returns. However, the C# compiler doesn't track field assignments through all helper methods. The compiler issues warning `CS8618` when fields aren't initialized directly in the constructor, but rather in a helper method. You add the `MemberNotNullAttribute` to a method declaration and specify the fields that are initialized to a non-null value in the method. For example, consider the following example:

```
public class Container
{
    private string _uniqueIdentifier; // must be initialized.
    private string? _optionalMessage;

    public Container()
    {
        Helper();
    }

    public Container(string message)
    {
        Helper();
        _optionalMessage = message;
    }

    [MemberNotNull(nameof(_uniqueIdentifier))]
    private void Helper()
    {
        _uniqueIdentifier = DateTime.Now.Ticks.ToString();
    }
}
```

You can specify multiple field names as arguments to the `MemberNotNull` attribute constructor.

The `MemberNotNullWhenAttribute` has a `bool` argument. You use `MemberNotNullWhen` in situations where your helper method returns a `bool` indicating whether your helper method initialized fields.

Stop nullable analysis when called method throws

Some methods, typically exception helpers or other utility methods, always exit by throwing an exception. Or, a helper may throw an exception based on the value of a Boolean argument.

In the first case, you can add the [DoesNotReturnAttribute](#) attribute to the method declaration. The compiler's *null-state* analysis doesn't check any code in a method that follows a call to a method annotated with

`DoesNotReturn`. Consider this method:

```
[DoesNotReturn]
private void FailFast()
{
    throw new InvalidOperationException();
}

public void SetState(object containedField)
{
    if (containedField is null)
    {
        FailFast();
    }

    // containedField can't be null:
    _field = containedField;
}
```

The compiler doesn't issue any warnings after the call to `FailFast`.

In the second case, you add the [System.Diagnostics.CodeAnalysis.DoesNotReturnIfAttribute](#) attribute to a Boolean parameter of the method. You can modify the previous example as follows:

```
private void FailFastIf([DoesNotReturnIf(true)] bool isNull)
{
    if (isNull)
    {
        throw new InvalidOperationException();
    }
}

public void SetFieldState(object? containedField)
{
    FailFastIf(containedField == null);
    // No warning: containedField can't be null here:
    _field = containedField;
}
```

When the value of the argument matches the value of the `DoesNotReturnIf` constructor, the compiler doesn't perform any *null-state* analysis after that method.

Summary

IMPORTANT

The official documentation tracks the latest C# version. We are currently writing for C# 9.0. Depending on the version of C# you're using, various features may not be available. The *default* C# version for your project is based on the target framework. For more information, see [C# language versioning defaults](#).

Adding nullable reference types provides an initial vocabulary to describe your APIs expectations for variables

that could be `null`. The attributes provide a richer vocabulary to describe the null state of variables as preconditions and postconditions. These attributes more clearly describe your expectations and provide a better experience for the developers using your APIs.

As you update libraries for a nullable context, add these attributes to guide users of your APIs to the correct usage. These attributes help you fully describe the null-state of arguments and return values.

- **AllowNull**: A non-nullable field, parameter, or property may be null.
- **DisallowNull**: A nullable field, parameter, or property should never be null.
- **MaybeNull**: A non-nullable field, parameter, property, or return value may be null.
- **NotNull**: A nullable field, parameter, property, or return value will never be null.
- **MaybeNullWhen**: A non-nullable argument may be null when the method returns the specified `bool` value.
- **NotNullWhen**: A nullable argument won't be null when the method returns the specified `bool` value.
- **NotNullIfNotNull**: A parameter, property, or return value isn't null if the argument for the specified parameter isn't null.
- **DoesNotReturn**: A method or property never returns. In other words, it always throws an exception.
- **DoesNotReturnIf**: This method or property never returns if the associated `bool` parameter has the specified value.

Miscellaneous attributes interpreted by the C# compiler

12/28/2021 • 10 minutes to read • [Edit Online](#)

The attributes `Conditional`, `Obsolete`, `AttributeUsage`, `AsyncMethodBuilder`, `InterpolatedStringHandler`, and `ModuleInitializer` can be applied to elements in your code. They add semantic meaning to those elements. The compiler uses those semantic meanings to alter its output and report possible mistakes by developers using your code.

`Conditional` attribute

The `Conditional` attribute makes the execution of a method dependent on a preprocessing identifier. The `Conditional` attribute is an alias for `ConditionalAttribute`, and can be applied to a method or an attribute class.

In the following example, `Conditional` is applied to a method to enable or disable the display of program-specific diagnostic information:

```
#define TRACE_ON
using System;
using System.Diagnostics;

namespace AttributeExamples
{
    public class Trace
    {
        [Conditional("TRACE_ON")]
        public static void Msg(string msg)
        {
            Console.WriteLine(msg);
        }
    }

    public class TraceExample
    {
        public static void Main()
        {
            Trace.Msg("Now in Main...");
            Console.WriteLine("Done.");
        }
    }
}
```

If the `TRACE_ON` identifier isn't defined, the trace output isn't displayed. Explore for yourself in the interactive window.

The `Conditional` attribute is often used with the `DEBUG` identifier to enable trace and logging features for debug builds but not in release builds, as shown in the following example:

```
[Conditional("DEBUG")]
static void DebugMethod()
{
}
```

When a method marked conditional is called, the presence or absence of the specified preprocessing symbol

determines whether the compiler includes or omits calls to the method. If the symbol is defined, the call is included; otherwise, the call is omitted. A conditional method must be a method in a class or struct declaration and must have a `void` return type. Using `Conditional` is cleaner, more elegant, and less error-prone than enclosing methods inside `#if...#endif` blocks.

If a method has multiple `Conditional` attributes, compiler includes calls to the method if one or more conditional symbols are defined (the symbols are logically linked together by using the OR operator). In the following example, the presence of either `A` or `B` results in a method call:

```
[Conditional("A"), Conditional("B")]
static void DoIfAorB()
{
    // ...
}
```

Using `Conditional` with attribute classes

The `Conditional` attribute can also be applied to an attribute class definition. In the following example, the custom attribute `Documentation` will only add information to the metadata if `DEBUG` is defined.

```
[Conditional("DEBUG")]
public class DocumentationAttribute : System.Attribute
{
    string text;

    public DocumentationAttribute(string text)
    {
        this.text = text;
    }
}

class SampleClass
{
    // This attribute will only be included if DEBUG is defined.
    [Documentation("This method displays an integer.")]
    static void DoWork(int i)
    {
        System.Console.WriteLine(i.ToString());
    }
}
```

obsolete attribute

The `obsolete` attribute marks a code element as no longer recommended for use. Use of an entity marked obsolete generates a warning or an error. The `obsolete` attribute is a single-use attribute and can be applied to any entity that allows attributes. `Obsolete` is an alias for [ObsoleteAttribute](#).

In the following example, the `obsolete` attribute is applied to class `A` and to method `B.OldMethod`. Because the second argument of the attribute constructor applied to `B.OldMethod` is set to `true`, this method will cause a compiler error, whereas using class `A` will just produce a warning. Calling `B.NewMethod`, however, produces no warning or error. For example, when you use it with the previous definitions, the following code generates two warnings and one error:

```

using System;

namespace AttributeExamples
{
    [Obsolete("use class B")]
    public class A
    {
        public void Method() { }
    }

    public class B
    {
        [Obsolete("use NewMethod", true)]
        public void OldMethod() { }

        public void NewMethod() { }
    }

    public static class ObsoleteProgram
    {
        public static void Main()
        {
            // Generates 2 warnings:
            A a = new A();

            // Generate no errors or warnings:
            B b = new B();
            b.NewMethod();

            // Generates an error, compilation fails.
            // b.OldMethod();
        }
    }
}

```

The string provided as the first argument to the attribute constructor will be displayed as part of the warning or error. Two warnings for class `A` are generated: one for the declaration of the class reference, and one for the class constructor. The `Obsolete` attribute can be used without arguments, but including an explanation what to use instead is recommended.

In C# 10, you can use constant string interpolation and the `nameof` operator to ensure the names match:

```

public class B
{
    [Obsolete($"use {nameof(NewMethod)} instead", true)]
    public void OldMethod() { }

    public void NewMethod() { }
}

```

AttributeUsage attribute

The `AttributeUsage` attribute determines how a custom attribute class can be used. [AttributeUsageAttribute](#) is an attribute you apply to custom attribute definitions. The `AttributeUsage` attribute enables you to control:

- Which program elements attribute may be applied to. Unless you restrict its usage, an attribute may be applied to any of the following program elements:
 - Assembly
 - Module
 - Field

- Event
- Method
- Param
- Property
- Return
- Type
- Whether an attribute can be applied to a single program element multiple times.
- Whether attributes are inherited by derived classes.

The default settings look like the following example when applied explicitly:

```
[AttributeUsage(AttributeTargets.All,
                AllowMultiple = false,
                Inherited = true)]
class NewAttribute : Attribute { }
```

In this example, the `NewAttribute` class can be applied to any supported program element. But it can be applied only once to each entity. The attribute is inherited by derived classes when applied to a base class.

The `AllowMultiple` and `Inherited` arguments are optional, so the following code has the same effect:

```
[AttributeUsage(AttributeTargets.All)]
class NewAttribute : Attribute { }
```

The first `AttributeUsageAttribute` argument must be one or more elements of the `AttributeTargets` enumeration. Multiple target types can be linked together with the OR operator, like the following example shows:

```
[AttributeUsage(AttributeTargets.Property | AttributeTargets.Field)]
class NewPropertyOrFieldAttribute : Attribute { }
```

Beginning in C# 7.3, attributes can be applied to either the property or the backing field for an auto-implemented property. The attribute applies to the property, unless you specify the `field` specifier on the attribute. Both are shown in the following example:

```
class MyClass
{
    // Attribute attached to property:
    [NewPropertyOrField]
    public string Name { get; set; } = string.Empty;

    // Attribute attached to backing field:
    [field: NewPropertyOrField]
    public string Description { get; set; } = string.Empty;
}
```

If the `AllowMultiple` argument is `true`, then the resulting attribute can be applied more than once to a single entity, as shown in the following example:

```
[AttributeUsage(AttributeTargets.Class, AllowMultiple = true)]
class MultiUse : Attribute { }

[MultiUse]
[MultiUse]
class Class1 { }

[MultiUse, MultiUse]
class Class2 { }
```

In this case, `MultiUseAttribute` can be applied repeatedly because `AllowMultiple` is set to `true`. Both formats shown for applying multiple attributes are valid.

If `Inherited` is `false`, then the attribute isn't inherited by classes derived from an attributed class. For example:

```
[AttributeUsage(AttributeTargets.Class, Inherited = false)]
class NonInheritedAttribute : Attribute { }

[NonInherited]
class BClass { }

class DClass : BClass { }
```

In this case `NonInheritedAttribute` isn't applied to `DClass` via inheritance.

You can also use these keywords to specify where an attribute should be applied. For example, you can use the `field:` specifier to add an attribute to the backing field of an [auto-implemented property](#). Or you can use the `field:`, `property:` or `param:` specifier to apply an attribute to any of the elements generated from a positional record. For an example, see [Positional syntax for property definition](#).

`AsyncMethodBuilder` attribute

Beginning with C# 7, you add the [System.Runtime.CompilerServices.AsyncMethodBuilderAttribute](#) attribute to a type that can be an async return type. The attribute specifies the type that builds the async method implementation when the specified type is returned from an async method. The `AsyncMethodBuilder` attribute can be applied to a type that:

- Has an accessible `GetAwaiter` method.
- The object returned by the `GetAwaiter` method implements the [System.Runtime.CompilerServices.ICriticalNotifyCompletion](#) interface.

The constructor to the `AsyncMethodBuilder` attribute specifies the type of the associated builder. The builder must implement the following accessible members:

- A static `Create()` method that returns the type of the builder.
- A readable `Task` property that returns the async return type.
- A `void SetException(Exception)` method that sets the exception when a task faults.
- A `void SetResult()` or `void SetResult(T result)` method that marks the task as completed and optionally sets the task's result
- A `Start` method with the following API signature:


```
void Start<TStateMachine>(ref TStateMachine stateMachine)
    where TStateMachine : System.Runtime.CompilerServices.IAsyncStateMachine
```

- An `AwaitOnCompleted` method with the following signature:

```
public void AwaitOnCompleted<TAwaiter, TStateMachine>(ref TAwaiter awaiter, ref TStateMachine stateMachine)
    where TAwaiter : System.Runtime.CompilerServices.INotifyCompletion
    where TStateMachine : System.Runtime.CompilerServices.IAsyncStateMachine
```

- An `AwaitUnsafeOnCompleted` method with the following signature:

```
public void AwaitUnsafeOnCompleted<TAwaiter, TStateMachine>(ref TAwaiter awaiter, ref TStateMachine stateMachine)
    where TAwaiter : System.Runtime.CompilerServices.ICriticalNotifyCompletion
    where TStateMachine : System.Runtime.CompilerServices.IAsyncStateMachine
```

You can learn about async method builders by reading about the following builders supplied by .NET:

- [System.Runtime.CompilerServices.AsyncTaskMethodBuilder](#)
- [System.Runtime.CompilerServices.AsyncTaskMethodBuilder<TResult>](#)
- [System.Runtime.CompilerServices.AsyncValueTaskMethodBuilder](#)
- [System.Runtime.CompilerServices.AsyncValueTaskMethodBuilder<TResult>](#)

In C# 10 and later, the `AsyncMethodBuilder` attribute can be applied to an async method to override the builder for that type.

`InterpolatedStringHandler` and `InterpolatedStringHandlerArguments` attributes

Starting with C# 10, you use these attributes to specify that a type is an *interpolated string handler*. The .NET 6 library already includes [System.Runtime.CompilerServices.DefaultInterpolatedStringHandler](#) for scenarios where you use an interpolated string as the argument for a `string` parameter. You may have other instances where you want to control how interpolated strings are processed. You apply the [System.Runtime.CompilerServices.InterpolatedStringHandlerAttribute](#) to the type that implements your handler. You apply the [System.Runtime.CompilerServices.InterpolatedStringHandlerArgumentAttribute](#) to parameters of that type's constructor.

You can learn more about building an interpolated string handler in the C# 10 feature specification for [interpolated string improvements](#).

`ModuleInitializer` attribute

Starting with C# 9, the `ModuleInitializer` attribute marks a method that the runtime calls when the assembly loads. `ModuleInitializer` is an alias for [ModuleInitializerAttribute](#).

The `ModuleInitializer` attribute can only be applied to a method that:

- Is static.
- Is parameterless.
- Returns `void`.
- Is accessible from the containing module, that is, `internal` or `public`.

- Isn't a generic method.
- Isn't contained in a generic class.
- Isn't a local function.

The `ModuleInitializer` attribute can be applied to multiple methods. In that case, the order in which the runtime calls them is deterministic but not specified.

The following example illustrates use of multiple module initializer methods. The `Init1` and `Init2` methods run before `Main`, and each adds a string to the `Text` property. So when `Main` runs, the `Text` property already has strings from both initializer methods.

```
using System;

internal class ModuleInitializerExampleMain
{
    public static void Main()
    {
        Console.WriteLine(ModuleInitializerExampleModule.Text);
        //output: Hello from Init1! Hello from Init2!
    }
}
```

```
using System.Runtime.CompilerServices;

internal class ModuleInitializerExampleModule
{
    public static string? Text { get; set; }

    [ModuleInitializer]
    public static void Init1()
    {
        Text += "Hello from Init1! ";
    }

    [ModuleInitializer]
    public static void Init2()
    {
        Text += "Hello from Init2! ";
    }
}
```

Source code generators sometimes need to generate initialization code. Module initializers provide a standard place for that code. In most other cases, you should write a [static constructor](#) instead of a module initializer.

`SkipLocalsInit` attribute

Starting in C# 9, the `SkipLocalsInit` attribute prevents the compiler from setting the `.locals init` flag when emitting to metadata. The `SkipLocalsInit` attribute is a single-use attribute and can be applied to a method, a property, a class, a struct, an interface, or a module, but not to an assembly. `SkipLocalsInit` is an alias for [SkipLocalsInitAttribute](#).

The `.locals init` flag causes the CLR to initialize all of the local variables declared in a method to their default values. Since the compiler also makes sure that you never use a variable before assigning some value to it, `.locals init` is typically not necessary. However, the extra zero-initialization may have measurable performance impact in some scenarios, such as when you use [stackalloc](#) to allocate an array on the stack. In those cases, you can add the `SkipLocalsInit` attribute. If applied to a method directly, the attribute affects that method and all its nested functions, including lambdas and local functions. If applied to a type or module, it affects all methods nested inside. This attribute doesn't affect abstract methods, but it does affect code

generated for the implementation.

This attribute requires the [AllowUnsafeBlocks](#) compiler option. This requirement signals that in some cases code could view unassigned memory (for example, reading from uninitialized stack-allocated memory).

The following example illustrates the effect of `SkipLocalsInit` attribute on a method that uses `stackalloc`. The method displays whatever was in memory when the array of integers was allocated.

```
[SkipLocalsInit]
static void ReadUninitializedMemory()
{
    Span<int> numbers = stackalloc int[120];
    for (int i = 0; i < 120; i++)
    {
        Console.WriteLine(numbers[i]);
    }
}
// output depends on initial contents of memory, for example:
//0
//0
//0
//168
//0
//-1271631451
//32767
//38
//0
//0
//0
//38
// Remaining rows omitted for brevity.
```

To try this code yourself, set the `AllowUnsafeBlocks` compiler option in your `.csproj` file:

```
<PropertyGroup>
    ...
    <AllowUnsafeBlocks>true</AllowUnsafeBlocks>
</PropertyGroup>
```

See also

- [Attribute](#)
- [System.Reflection](#)
- [Attributes](#)
- [Reflection](#)

Unsafe code, pointer types, and function pointers

12/28/2021 • 13 minutes to read • [Edit Online](#)

Most of the C# code you write is "verifiably safe code." *Verifiably safe code* means .NET tools can verify that the code is safe. In general, safe code doesn't directly access memory using pointers. It also doesn't allocate raw memory. It creates managed objects instead.

C# supports an `unsafe` context, in which you may write *unverifiable* code. In an `unsafe` context, code may use pointers, allocate and free blocks of memory, and call methods using function pointers. Unsafe code in C# isn't necessarily dangerous; it's just code whose safety cannot be verified.

Unsafe code has the following properties:

- Methods, types, and code blocks can be defined as unsafe.
- In some cases, unsafe code may increase an application's performance by removing array bounds checks.
- Unsafe code is required when you call native functions that require pointers.
- Using unsafe code introduces security and stability risks.
- The code that contains unsafe blocks must be compiled with the `AllowUnsafeBlocks` compiler option.

Pointer types

In an unsafe context, a type may be a pointer type, in addition to a value type, or a reference type. A pointer type declaration takes one of the following forms:

```
type* identifier;  
void* identifier; //allowed but not recommended
```

The type specified before the `*` in a pointer type is called the **referent type**. Only an `unmanaged type` can be a referent type.

Pointer types don't inherit from `object` and no conversions exist between pointer types and `object`. Also, boxing and unboxing don't support pointers. However, you can convert between different pointer types and between pointer types and integral types.

When you declare multiple pointers in the same declaration, you write the asterisk (`*`) together with the underlying type only. It isn't used as a prefix to each pointer name. For example:

```
int* p1, p2, p3;    // Ok  
int *p1, *p2, *p3; // Invalid in C#
```

A pointer can't point to a reference or to a `struct` that contains references, because an object reference can be garbage collected even if a pointer is pointing to it. The garbage collector doesn't keep track of whether an object is being pointed to by any pointer types.

The value of the pointer variable of type `MyType*` is the address of a variable of type `MyType`. The following are examples of pointer type declarations:

- `int* p : p` is a pointer to an integer.
- `int** p : p` is a pointer to a pointer to an integer.
- `int*[] p : p` is a single-dimensional array of pointers to integers.

- `char* p`: `p` is a pointer to a char.
- `void* p`: `p` is a pointer to an unknown type.

The pointer indirection operator `*` can be used to access the contents at the location pointed to by the pointer variable. For example, consider the following declaration:

```
int* myVariable;
```

The expression `*myVariable` denotes the `int` variable found at the address contained in `myVariable`.

There are several examples of pointers in the articles on the [fixed statement](#). The following example uses the `unsafe` keyword and the `fixed` statement, and shows how to increment an interior pointer. You can paste this code into the Main function of a console application to run it. These examples must be compiled with the [AllowUnsafeBlocks](#) compiler option set.

```
// Normal pointer to an object.
int[] a = new int[5] { 10, 20, 30, 40, 50 };
// Must be in unsafe code to use interior pointers.
unsafe
{
    // Must pin object on heap so that it doesn't move while using interior pointers.
    fixed (int* p = &a[0])
    {
        // p is pinned as well as object, so create another pointer to show incrementing it.
        int* p2 = p;
        Console.WriteLine(*p2);
        // Incrementing p2 bumps the pointer by four bytes due to its type ...
        p2 += 1;
        Console.WriteLine(*p2);
        p2 += 1;
        Console.WriteLine(*p2);
        Console.WriteLine("-----");
        Console.WriteLine(*p);
        // Dereferencing p and incrementing changes the value of a[0] ...
        *p += 1;
        Console.WriteLine(*p);
        *p += 1;
        Console.WriteLine(*p);
    }
}

Console.WriteLine("-----");
Console.WriteLine(a[0]);

/*
Output:
10
20
30
-----
10
11
12
-----
12
*/
```

You can't apply the indirection operator to a pointer of type `void*`. However, you can use a cast to convert a void pointer to any other pointer type, and vice versa.

A pointer can be `null`. Applying the indirection operator to a null pointer causes an implementation-defined behavior.

Passing pointers between methods can cause undefined behavior. Consider a method that returns a pointer to a local variable through an `in`, `out`, or `ref` parameter or as the function result. If the pointer was set in a fixed block, the variable to which it points may no longer be fixed.

The following table lists the operators and statements that can operate on pointers in an unsafe context:

OPERATOR/STATEMENT	USE
<code>*</code>	Performs pointer indirection.
<code>-></code>	Accesses a member of a struct through a pointer.
<code>[]</code>	Indexes a pointer.
<code>&</code>	Obtains the address of a variable.
<code>++</code> and <code>--</code>	Increments and decrements pointers.
<code>+</code> and <code>-</code>	Performs pointer arithmetic.
<code>==</code> , <code>!=</code> , <code><</code> , <code>></code> , <code><=</code> , and <code>>=</code>	Compares pointers.
<code>stackalloc</code>	Allocates memory on the stack.
<code>fixed</code> statement	Temporarily fixes a variable so that its address may be found.

For more information about pointer-related operators, see [Pointer-related operators](#).

Any pointer type can be implicitly converted to a `void*` type. Any pointer type can be assigned the value `null`. Any pointer type can be explicitly converted to any other pointer type using a cast expression. You can also convert any integral type to a pointer type, or any pointer type to an integral type. These conversions require an explicit cast.

The following example converts an `int*` to a `byte*`. Notice that the pointer points to the lowest addressed byte of the variable. When you successively increment the result, up to the size of `int` (4 bytes), you can display the remaining bytes of the variable.

```

int number = 1024;

unsafe
{
    // Convert to byte:
    byte* p = (byte*)&number;

    System.Console.WriteLine("The 4 bytes of the integer:");

    // Display the 4 bytes of the int variable:
    for (int i = 0 ; i < sizeof(int) ; ++i)
    {
        System.Console.WriteLine(" {0:X2}", *p);
        // Increment the pointer:
        p++;
    }
    System.Console.WriteLine();
    System.Console.WriteLine("The value of the integer: {0}", number);

    /* Output:
        The 4 bytes of the integer: 00 04 00 00
        The value of the integer: 1024
    */
}

```

Fixed-size buffers

In C#, you can use the `fixed` statement to create a buffer with a fixed size array in a data structure. Fixed size buffers are useful when you write methods that interoperate with data sources from other languages or platforms. The fixed array can take any attributes or modifiers that are allowed for regular struct members. The only restriction is that the array type must be `bool`, `byte`, `char`, `short`, `int`, `long`, `sbyte`, `ushort`, `uint`, `ulong`, `float`, or `double`.

```
private fixed char name[30];
```

In safe code, a C# struct that contains an array doesn't contain the array elements. The struct contains a reference to the elements instead. You can embed an array of fixed size in a `struct` when it's used in an `unsafe` code block.

The size of the following `struct` doesn't depend on the number of elements in the array, since `pathName` is a reference:

```

public struct PathArray
{
    public char[] pathName;
    private int reserved;
}

```

A `struct` can contain an embedded array in unsafe code. In the following example, the `fixedBuffer` array has a fixed size. You use a `fixed` statement to establish a pointer to the first element. You access the elements of the array through this pointer. The `fixed` statement pins the `fixedBuffer` instance field to a specific location in memory.

```

internal unsafe struct Buffer
{
    public fixed char fixedBuffer[128];
}

internal unsafe class Example
{
    public Buffer buffer = default;
}

private static void AccessEmbeddedArray()
{
    var example = new Example();

    unsafe
    {
        // Pin the buffer to a fixed location in memory.
        fixed (char* charPtr = example.buffer.fixedBuffer)
        {
            *charPtr = 'A';
        }
        // Access safely through the index:
        char c = example.buffer.fixedBuffer[0];
        Console.WriteLine(c);

        // Modify through the index:
        example.buffer.fixedBuffer[0] = 'B';
        Console.WriteLine(example.buffer.fixedBuffer[0]);
    }
}

```

The size of the 128 element `char` array is 256 bytes. Fixed size `char` buffers always take 2 bytes per character, regardless of the encoding. This array size is the same even when char buffers are marshaled to API methods or structs with `CharSet = CharSet.Auto` or `CharSet = CharSet.Ansi`. For more information, see [CharSet](#).

The preceding example demonstrates accessing `fixed` fields without pinning, which is available starting with C# 7.3.

Another common fixed-size array is the `bool` array. The elements in a `bool` array are always 1 byte in size. `bool` arrays aren't appropriate for creating bit arrays or buffers.

Fixed size buffers are compiled with the [System.Runtime.CompilerServices.UnsafeValueTypeAttribute](#), which instructs the common language runtime (CLR) that a type contains an unmanaged array that can potentially overflow. Memory allocated using `stackalloc` also automatically enables buffer overrun detection features in the CLR. The previous example shows how a fixed size buffer could exist in an `unsafe struct`.

```

internal unsafe struct Buffer
{
    public fixed char fixedBuffer[128];
}

```

The compiler-generated C# for `Buffer` is attributed as follows:


```
internal struct Buffer
{
    [StructLayout(LayoutKind.Sequential, Size = 256)]
    [CompilerGenerated]
    [UnsafeValueType]
    public struct <fixedBuffer>e__FixedBuffer
    {
        public char FixedElementField;
    }

    [FixedBuffer(typeof(char), 128)]
    public <fixedBuffer>e__FixedBuffer fixedBuffer;
}
```

Fixed size buffers differ from regular arrays in the following ways:

- May only be used in an `unsafe` context.
- May only be instance fields of structs.
- They're always vectors, or one-dimensional arrays.
- The declaration should include the length, such as `fixed char id[8]`. You can't use `fixed char id[]`.

How to use pointers to copy an array of bytes

The following example uses pointers to copy bytes from one array to another.

This example uses the `unsafe` keyword, which enables you to use pointers in the `Copy` method. The `fixed` statement is used to declare pointers to the source and destination arrays. The `fixed` statement *pins* the location of the source and destination arrays in memory so that they will not be moved by garbage collection. The memory blocks for the arrays are unpinned when the `fixed` block is completed. Because the `Copy` method in this example uses the `unsafe` keyword, it must be compiled with the `AllowUnsafeBlocks` compiler option.

This example accesses the elements of both arrays using indices rather than a second unmanaged pointer. The declaration of the `pSource` and `pTarget` pointers pins the arrays. This feature is available starting with C# 7.3.

```
static unsafe void Copy(byte[] source, int sourceOffset, byte[] target,
    int targetOffset, int count)
{
    // If either array is not instantiated, you cannot complete the copy.
    if ((source == null) || (target == null))
    {
        throw new System.ArgumentException();
    }

    // If either offset, or the number of bytes to copy, is negative, you
    // cannot complete the copy.
    if ((sourceOffset < 0) || (targetOffset < 0) || (count < 0))
    {
        throw new System.ArgumentException();
    }

    // If the number of bytes from the offset to the end of the array is
    // less than the number of bytes you want to copy, you cannot complete
    // the copy.
    if ((source.Length - sourceOffset < count) ||
        (target.Length - targetOffset < count))
    {
        throw new System.ArgumentException();
    }

    // The following fixed statement pins the location of the source and
    // target objects in memory so that they will not be moved by garbage
    // collection.
```

```

// collection.
fixed (byte* pSource = source, pTarget = target)
{
    // Copy the specified number of bytes from source to target.
    for (int i = 0; i < count; i++)
    {
        pTarget[targetOffset + i] = pSource[sourceOffset + i];
    }
}

static void UnsafeCopyArrays()
{
    // Create two arrays of the same length.
    int length = 100;
    byte[] byteArray1 = new byte[length];
    byte[] byteArray2 = new byte[length];

    // Fill byteArray1 with 0 - 99.
    for (int i = 0; i < length; ++i)
    {
        byteArray1[i] = (byte)i;
    }

    // Display the first 10 elements in byteArray1.
    System.Console.WriteLine("The first 10 elements of the original are:");
    for (int i = 0; i < 10; ++i)
    {
        System.Console.Write(byteArray1[i] + " ");
    }
    System.Console.WriteLine("\n");

    // Copy the contents of byteArray1 to byteArray2.
    Copy(byteArray1, 0, byteArray2, 0, length);

    // Display the first 10 elements in the copy, byteArray2.
    System.Console.WriteLine("The first 10 elements of the copy are:");
    for (int i = 0; i < 10; ++i)
    {
        System.Console.Write(byteArray2[i] + " ");
    }
    System.Console.WriteLine("\n");

    // Copy the contents of the last 10 elements of byteArray1 to the
    // beginning of byteArray2.
    // The offset specifies where the copying begins in the source array.
    int offset = length - 10;
    Copy(byteArray1, offset, byteArray2, 0, length - offset);

    // Display the first 10 elements in the copy, byteArray2.
    System.Console.WriteLine("The first 10 elements of the copy are:");
    for (int i = 0; i < 10; ++i)
    {
        System.Console.Write(byteArray2[i] + " ");
    }
    System.Console.WriteLine("\n");
    /* Output:
        The first 10 elements of the original are:
        0 1 2 3 4 5 6 7 8 9

        The first 10 elements of the copy are:
        0 1 2 3 4 5 6 7 8 9

        The first 10 elements of the copy are:
        90 91 92 93 94 95 96 97 98 99
    */
}

```

Function pointers

C# provides `delegate` types to define safe function pointer objects. Invoking a delegate involves instantiating a type derived from `System.Delegate` and making a virtual method call to its `Invoke` method. This virtual call uses the `callvirt` IL instruction. In performance critical code paths, using the `calli` IL instruction is more efficient.

You can define a function pointer using the `delegate*` syntax. The compiler will call the function using the `calli` instruction rather than instantiating a `delegate` object and calling `Invoke`. The following code declares two methods that use a `delegate` or a `delegate*` to combine two objects of the same type. The first method uses a `System.Func<T1,T2,TResult>` delegate type. The second method uses a `delegate*` declaration with the same parameters and return type:

```
public static T Combine<T>(Func<T, T, T> combinator, T left, T right) =>
    combinator(left, right);

public static T UnsafeCombine<T>(delegate*<T, T, T> combinator, T left, T right) =>
    combinator(left, right);
```

The following code shows how you would declare a static local function and invoke the `UnsafeCombine` method using a pointer to that local function:

```
static int localMultiply(int x, int y) => x * y;
int product = UnsafeCombine(&localMultiply, 3, 4);
```

The preceding code illustrates several of the rules on the function accessed as a function pointer:

- Function pointers can only be declared in an `unsafe` context.
- Methods that take a `delegate*` (or return a `delegate*`) can only be called in an `unsafe` context.
- The `&` operator to obtain the address of a function is allowed only on `static` functions. (This rule applies to both member functions and local functions).

The syntax has parallels with declaring `delegate` types and using pointers. The `*` suffix on `delegate` indicates the declaration is a *function pointer*. The `&` when assigning a method group to a function pointer indicates the operation takes the address of the method.

You can specify the calling convention for a `delegate*` using the keywords `managed` and `unmanaged`. In addition, for `unmanaged` function pointers, you can specify the calling convention. The following declarations show examples of each. The first declaration uses the `managed` calling convention, which is the default. The next three use an `unmanaged` calling convention. Each specifies one of the ECMA 335 calling conventions: `Cdecl`, `Stdcall`, `Fastcall`, or `Thiscall`. The last declaration uses the `unmanaged` calling convention, instructing the CLR to pick the default calling convention for the platform. The CLR will choose the calling convention at run time.

```
public static T ManagedCombine<T>(delegate* managed<T, T, T> combinator, T left, T right) =>
    combinator(left, right);
public static T CDeclCombine<T>(delegate* unmanaged[Cdecl]<T, T, T> combinator, T left, T right) =>
    combinator(left, right);
public static T StdcallCombine<T>(delegate* unmanaged[Stdcall]<T, T, T> combinator, T left, T right) =>
    combinator(left, right);
public static T FastcallCombine<T>(delegate* unmanaged[Fastcall]<T, T, T> combinator, T left, T right) =>
    combinator(left, right);
public static T ThiscallCombine<T>(delegate* unmanaged[Thiscall]<T, T, T> combinator, T left, T right) =>
    combinator(left, right);
public static T UnmanagedCombine<T>(delegate* unmanaged<T, T, T> combinator, T left, T right) =>
    combinator(left, right);
```

You can learn more about function pointers in the [Function pointer](#) proposal for C# 9.0.

C# language specification

For more information, see the [Unsafe code](#) chapter of the [C# language specification](#).

C# preprocessor directives

12/28/2021 • 13 minutes to read • [Edit Online](#)

Although the compiler doesn't have a separate preprocessor, the directives described in this section are processed as if there were one. You use them to help in conditional compilation. Unlike C and C++ directives, you can't use these directives to create macros. A preprocessor directive must be the only instruction on a line.

Nullable context

The `#nullable` preprocessor directive sets the *nullable annotation context* and *nullable warning context*. This directive controls whether nullable annotations have effect, and whether nullability warnings are given. Each context is either *disabled* or *enabled*.

Both contexts can be specified at the project level (outside of C# source code). The `#nullable` directive controls the annotation and warning contexts and takes precedence over the project-level settings. A directive sets the context(s) it controls until another directive overrides it, or until the end of the source file.

The effect of the directives is as follows:

- `#nullable disable` : Sets the nullable annotation and warning contexts to *disabled*.
- `#nullable enable` : Sets the nullable annotation and warning contexts to *enabled*.
- `#nullable restore` : Restores the nullable annotation and warning contexts to project settings.
- `#nullable disable annotations` : Sets the nullable annotation context to *disabled*.
- `#nullable enable annotations` : Sets the nullable annotation context to *enabled*.
- `#nullable restore annotations` : Restores the nullable annotation context to project settings.
- `#nullable disable warnings` : Sets the nullable warning context to *disabled*.
- `#nullable enable warnings` : Sets the nullable warning context to *enabled*.
- `#nullable restore warnings` : Restores the nullable warning context to project settings.

Conditional compilation

You use four preprocessor directives to control conditional compilation:

- `#if` : Opens a conditional compilation, where code is compiled only if the specified symbol is defined.
- `#elif` : Closes the preceding conditional compilation and opens a new conditional compilation based on if the specified symbol is defined.
- `#else` : Closes the preceding conditional compilation and opens a new conditional compilation if the previous specified symbol isn't defined.
- `#endif` : Closes the preceding conditional compilation.

When the C# compiler finds an `#if` directive, followed eventually by an `#endif` directive, it compiles the code between the directives only if the specified symbol is defined. Unlike C and C++, you can't assign a numeric value to a symbol. The `#if` statement in C# is Boolean and only tests whether the symbol has been defined or not. For example:

```
#if DEBUG
    Console.WriteLine("Debug version");
#endif
```

You can use the operators `==` (equality) and `!=` (inequality) to test for the `bool` values `true` or `false`. `true` means the symbol is defined. The statement `#if DEBUG` has the same meaning as `#if (DEBUG == true)`. You can use the `&&` (and), `||` (or), and `!` (not) operators to evaluate whether multiple symbols have been defined. You can also group symbols and operators with parentheses.

`#if`, along with the `#else`, `#elif`, `#endif`, `#define`, and `#undef` directives, lets you include or exclude code based on the existence of one or more symbols. Conditional compilation can be useful when compiling code for a debug build or when compiling for a specific configuration.

A conditional directive beginning with an `#if` directive must explicitly be terminated with an `#endif` directive. `#define` lets you define a symbol. By using the symbol as the expression passed to the `#if` directive, the expression evaluates to `true`. You can also define a symbol with the `DefineConstants` compiler option. You can undefine a symbol with `#undef`. The scope of a symbol created with `#define` is the file in which it was defined. A symbol that you define with `DefineConstants` or with `#define` doesn't conflict with a variable of the same name. That is, a variable name shouldn't be passed to a preprocessor directive, and a symbol can only be evaluated by a preprocessor directive.

`#elif` lets you create a compound conditional directive. The `#elif` expression will be evaluated if neither the preceding `#if` nor any preceding, optional, `#elif` directive expressions evaluate to `true`. If an `#elif` expression evaluates to `true`, the compiler evaluates all the code between the `#elif` and the next conditional directive. For example:

```
#define VC7
//...
#if debug
    Console.WriteLine("Debug build");
#elif VC7
    Console.WriteLine("Visual Studio 7");
#endif
```

`#else` lets you create a compound conditional directive, so that, if none of the expressions in the preceding `#if` or (optional) `#elif` directives evaluate to `true`, the compiler will evaluate all code between `#else` and the next `#endif`. `#endif` (#endif) must be the next preprocessor directive after `#else`.

`#endif` specifies the end of a conditional directive, which began with the `#if` directive.

The build system is also aware of predefined preprocessor symbols representing different `target frameworks` in SDK-style projects. They're useful when creating applications that can target more than one .NET version.

TARGET FRAMEWORKS	SYMBOLS	ADDITIONAL SYMBOLS AVAILABLE IN .NET 5+ SDK
.NET Framework	NETFRAMEWORK , NET48 , NET472 , NET471 , NET47 , NET462 , NET461 , NET46 , NET452 , NET451 , NET45 , NET40 , NET35 , NET20	NET48_OR_GREATER , NET472_OR_GREATER , NET471_OR_GREATER , NET47_OR_GREATER , NET462_OR_GREATER , NET461_OR_GREATER , NET46_OR_GREATER , NET452_OR_GREATER , NET451_OR_GREATER , NET45_OR_GREATER , NET40_OR_GREATER , NET35_OR_GREATER , NET20_OR_GREATER

TARGET FRAMEWORKS	SYMBOLS	ADDITIONAL SYMBOLS AVAILABLE IN .NET 5+ SDK
.NET Standard	NETSTANDARD , NETSTANDARD2_1 , NETSTANDARD2_0 , NETSTANDARD1_6 , NETSTANDARD1_5 , NETSTANDARD1_4 , NETSTANDARD1_3 , NETSTANDARD1_2 , NETSTANDARD1_1 , NETSTANDARD1_0	NETSTANDARD2_1_OR_GREATER , NETSTANDARD2_0_OR_GREATER , NETSTANDARD1_6_OR_GREATER , NETSTANDARD1_5_OR_GREATER , NETSTANDARD1_4_OR_GREATER , NETSTANDARD1_3_OR_GREATER , NETSTANDARD1_2_OR_GREATER , NETSTANDARD1_1_OR_GREATER , NETSTANDARD1_0_OR_GREATER
.NET 5+ (and .NET Core)	NET , NET6_0 , NET6_0_ANDROID , NET6_0_IOS , NET6_0_MACOS , NET6_0_MACCATALYST , NET6_0_TVOS , NET6_0_WINDOWS , NET5_0 , NETCOREAPP , NETCOREAPP3_1 , NETCOREAPP3_0 , NETCOREAPP2_2 , NETCOREAPP2_1 , NETCOREAPP2_0 , NETCOREAPP1_1 , NETCOREAPP1_0	NET6_0_OR_GREATER , NET6_0_ANDROID_OR_GREATER , NET6_0_IOS_OR_GREATER , NET6_0_MACOS_OR_GREATER , NET6_0_MACCATALYST_OR_GREATER , NET6_0_TVOS_OR_GREATER , NET6_0_WINDOWS_OR_GREATER , NET5_0_OR_GREATER , NETCOREAPP_OR_GREATER , NETCOREAPP3_1_OR_GREATER , NETCOREAPP3_0_OR_GREATER , NETCOREAPP2_2_OR_GREATER , NETCOREAPP2_1_OR_GREATER , NETCOREAPP2_0_OR_GREATER , NETCOREAPP1_1_OR_GREATER , NETCOREAPP1_0_OR_GREATER

NOTE

- Versionless symbols are defined regardless of the version you're targeting.
- Version-specific symbols are only defined for the version you're targeting.
- The `<framework>_OR_GREATER` symbols are defined for the version you're targeting and all earlier versions. For example, if you're targeting .NET Framework 2.0, the following symbols are defined: `NET_2_0` , `NET_2_0_OR_GREATER` , `NET_1_1_OR_GREATER` , and `NET_1_0_OR_GREATER` .

NOTE

For traditional, non-SDK-style projects, you have to manually configure the conditional compilation symbols for the different target frameworks in Visual Studio via the project's properties pages.

Other predefined symbols include the `DEBUG` and `TRACE` constants. You can override the values set for the project using `#define` . The `DEBUG` symbol, for example, is automatically set depending on your build configuration properties ("Debug" or "Release" mode).

The following example shows you how to define a `MYTEST` symbol on a file and then test the values of the `MYTEST` and `DEBUG` symbols. The output of this example depends on whether you built the project on **Debug** or **Release** configuration mode.

```
#define MYTEST
using System;
public class MyClass
{
    static void Main()
    {
#if (DEBUG && !MYTEST)
        Console.WriteLine("DEBUG is defined");
#elif (!DEBUG && MYTEST)
        Console.WriteLine("MYTEST is defined");
#elif (DEBUG && MYTEST)
        Console.WriteLine("DEBUG and MYTEST are defined");
#else
        Console.WriteLine("DEBUG and MYTEST are not defined");
#endif
    }
}
```

The following example shows you how to test for different target frameworks so you can use newer APIs when possible:

```
public class MyClass
{
    static void Main()
    {
#if NET40
        WebClient _client = new WebClient();
#else
        HttpClient _client = new HttpClient();
#endif
    }
    //...
}
```

Defining symbols

You use the following two preprocessor directives to define or undefine symbols for conditional compilation:

- `#define`: Define a symbol.
- `#undef`: Undefine a symbol.

You use `#define` to define a symbol. When you use the symbol as the expression that's passed to the `#if` directive, the expression will evaluate to `true`, as the following example shows:

```
#define VERBOSE

#if VERBOSE
    Console.WriteLine("Verbose output version");
#endif
```

NOTE

The `#define` directive cannot be used to declare constant values as is typically done in C and C++. Constants in C# are best defined as static members of a class or struct. If you have several such constants, consider creating a separate "Constants" class to hold them.

Symbols can be used to specify conditions for compilation. You can test for the symbol with either `#if` or

`#elif`. You can also use the [ConditionalAttribute](#) to perform conditional compilation. You can define a symbol, but you can't assign a value to a symbol. The `#define` directive must appear in the file before you use any instructions that aren't also preprocessor directives. You can also define a symbol with the [DefineConstants](#) compiler option. You can undefine a symbol with `#undef`.

Defining regions

You can define regions of code that can be collapsed in an outline using the following two preprocessor directives:

- `#region`: Start a region.
- `#endregion`: End a region.

`#region` lets you specify a block of code that you can expand or collapse when using the [outlining](#) feature of the code editor. In longer code files, it's convenient to collapse or hide one or more regions so that you can focus on the part of the file that you're currently working on. The following example shows how to define a region:

```
#region MyClass definition
public class MyClass
{
    static void Main()
    {
    }
}
#endregion
```

A `#region` block must be terminated with an `#endregion` directive. A `#region` block can't overlap with an `#if` block. However, a `#region` block can be nested in an `#if` block, and an `#if` block can be nested in a `#region` block.

Error and warning information

You instruct the compiler to generate user-defined compiler errors and warnings, and control line information using the following directives:

- `#error`: Generate a compiler error with a specified message.
- `#warning`: Generate a compiler warning, with a specific message.
- `#line`: Change the line number printed with compiler messages.

`#error` lets you generate a [CS1029](#) user-defined error from a specific location in your code. For example:

```
#error Deprecated code in this method.
```

NOTE

The compiler treats `#error version` in a special way and reports a compiler error, CS8304, with a message containing the used compiler and language versions.

`#warning` lets you generate a [CS1030](#) level one compiler warning from a specific location in your code. For example:

```
#warning Deprecated code in this method.
```

`#line` lets you modify the compiler's line numbering and (optionally) the file name output for errors and warnings.

The following example shows how to report two warnings associated with line numbers. The `#line 200` directive forces the next line's number to be 200 (although the default is #6), and until the next `#line` directive, the filename will be reported as "Special". The `#line default` directive returns the line numbering to its default numbering, which counts the lines that were renumbered by the previous directive.

```
class MainClass
{
    static void Main()
    {
#line 200 "Special"
        int i;
        int j;
#line default
        char c;
        float f;
#line hidden // numbering not affected
        string s;
        double d;
    }
}
```

Compilation produces the following output:

```
Special(200,13): warning CS0168: The variable 'i' is declared but never used
Special(201,13): warning CS0168: The variable 'j' is declared but never used
MainClass.cs(9,14): warning CS0168: The variable 'c' is declared but never used
MainClass.cs(10,15): warning CS0168: The variable 'f' is declared but never used
MainClass.cs(12,16): warning CS0168: The variable 's' is declared but never used
MainClass.cs(13,16): warning CS0168: The variable 'd' is declared but never used
```

The `#line` directive might be used in an automated, intermediate step in the build process. For example, if lines were removed from the original source code file, but you still wanted the compiler to generate output based on the original line numbering in the file, you could remove lines and then simulate the original line numbering with `#line`.

The `#line hidden` directive hides the successive lines from the debugger, such that when the developer steps through the code, any lines between a `#line hidden` and the next `#line` directive (assuming that it isn't another `#line hidden` directive) will be stepped over. This option can also be used to allow ASP.NET to differentiate between user-defined and machine-generated code. Although ASP.NET is the primary consumer of this feature, it's likely that more source generators will make use of it.

A `#line hidden` directive doesn't affect file names or line numbers in error reporting. That is, if the compiler finds an error in a hidden block, the compiler will report the current file name and line number of the error.

The `#line filename` directive specifies the file name you want to appear in the compiler output. By default, the actual name of the source code file is used. The file name must be in double quotation marks (") and must be preceded by a line number.

Beginning with C# 10, you can use a new form of the `#line` directive:

```
#line (1, 1) - (5, 60) 10 "partial-class.g.cs"
/*34567*/int b = 0;
```

The components of this form are:

- `(1, 1)` : The start line and column for the first character on the line that follows the directive. In this example, the next line would be reported as line 1, column 1.
- `(5, 60)` : The end line and column for the marked region.
- `10` : The column offset for the `#line` directive to take effect. In this example, the 10th column would be reported as column one. That's where the declaration `int b = 0;` begins. This field is optional. If omitted, the directive takes effect on the first column.
- `"partial-class.g.cs"` : The name of the output file.

The preceding example would generate the following warning:

```
partial-class.g.cs(1,5,1,6): warning CS0219: The variable 'b' is assigned but its value is never used
```

After remapping, the variable, `b`, is on the first line, at character six.

Domain-specific languages (DSLs) typically use this format to provide a better mapping from the source file to the generated C# output. To see more examples of this format, see the [feature specification](#) in the section on examples.

Pragmas

`#pragma` gives the compiler special instructions for the compilation of the file in which it appears. The instructions must be supported by the compiler. In other words, you can't use `#pragma` to create custom preprocessing instructions.

- `#pragma warning` : Enable or disable warnings.
- `#pragma checksum` : Generate a checksum.

```
#pragma pragma-name pragma-arguments
```

Where `pragma-name` is the name of a recognized pragma and `pragma-arguments` is the pragma-specific arguments.

#pragma warning

`#pragma warning` can enable or disable certain warnings.

```
#pragma warning disable warning-list
#pragma warning restore warning-list
```

Where `warning-list` is a comma-separated list of warning numbers. The "CS" prefix is optional. When no warning numbers are specified, `disable` disables all warnings and `restore` enables all warnings.

NOTE

To find warning numbers in Visual Studio, build your project and then look for the warning numbers in the **Output** window.

The `disable` takes effect beginning on the next line of the source file. The warning is restored on the line following the `restore`. If there's no `restore` in the file, the warnings are restored to their default state at the first line of any later files in the same compilation.

```
// pragma_warning.cs
using System;

#pragma warning disable 414, CS3021
[CLSCompliant(false)]
public class C
{
    int i = 1;
    static void Main()
    {
    }
}

#pragma warning restore CS3021
[CLSCompliant(false)] // CS3021
public class D
{
    int i = 1;
    public static void F()
    {
    }
}
```

#pragma checksum

Generates checksums for source files to aid with debugging ASP.NET pages.

```
#pragma checksum "filename" "{guid}" "checksum bytes"
```

Where "filename" is the name of the file that requires monitoring for changes or updates, "{guid}" is the Globally Unique Identifier (GUID) for the hash algorithm, and "checksum_bytes" is the string of hexadecimal digits representing the bytes of the checksum. Must be an even number of hexadecimal digits. An odd number of digits results in a compile-time warning, and the directive is ignored.

The Visual Studio debugger uses a checksum to make sure that it always finds the right source. The compiler computes the checksum for a source file, and then emits the output to the program database (PDB) file. The debugger then uses the PDB to compare against the checksum that it computes for the source file.

This solution doesn't work for ASP.NET projects, because the computed checksum is for the generated source file, rather than the .aspx file. To address this problem, `#pragma checksum` provides checksum support for ASP.NET pages.

When you create an ASP.NET project in Visual C#, the generated source file contains a checksum for the .aspx file, from which the source is generated. The compiler then writes this information into the PDB file.

If the compiler doesn't find a `#pragma checksum` directive in the file, it computes the checksum and writes the value to the PDB file.

```
class TestClass
{
    static int Main()
    {
        #pragma checksum "file.cs" "{406EA660-64CF-4C82-B6F0-42D48172A799}" "ab007f1d23d9" // New checksum
    }
}
```

C# compiler options

12/28/2021 • 2 minutes to read • [Edit Online](#)

This section describes the options interpreted by the C# compiler. Options are grouped into separate articles based on what they control, for example, language features, code generation, and output. Use the table of contents to navigate amongst them.

How to set options

There are two different ways to set compiler options in .NET projects:

- ***In your *.csproj file***

You can add MSBuild properties for any compiler option in your *.csproj file in XML format. The property name is the same as the compiler option. The value of the property sets the value of the compiler option. For example, the following project file snippet sets the `LangVersion` property.

```
<PropertyGroup>
  <LangVersion>preview</LangVersion>
</PropertyGroup>
```

For more information on setting options in project files, see the article [MSBuild properties for .NET SDK Projects](#).

- ***Using the Visual Studio Property pages***

Visual Studio provides property pages to edit build properties. To learn more about them, see [Manage project and solution properties - Windows](#) or [Manage project and solution properties - Mac](#).

.NET Framework projects

IMPORTANT

This section applies to .NET Framework projects only.

In addition to the mechanisms described above, you can set compiler options using two additional methods for .NET Framework projects:

- **Command line arguments for .NET Framework projects:** .NET Framework projects use `csc.exe` instead of `dotnet build` to build projects. You can specify command line arguments to `csc.exe` for .NET Framework projects.
- **Compiled ASP.NET pages:** .NET Framework projects use a section of the `web.config` file for compiling pages. For the new build system, and ASP.NET Core projects, options are taken from the project file.

The word for some compiler options changed from `csc.exe` and .NET Framework projects to the new MSBuild system. The new syntax is used throughout this section. Both versions are listed at the top of each page. For `csc.exe`, any arguments are listed following the option and a colon. For example, the `-doc` option would be:

```
-doc:DocFile.xml
```

You can invoke the C# compiler by typing the name of its executable file (`csc.exe`) at a command prompt.

For .NET Framework projects, you can also run *csc.exe* from the command line. Every compiler option is available in two forms: **-option** and **/option**. In .NET Framework web projects, you specify options for compiling code-behind in the *web.config* file. For more information, see [<compiler> Element](#).

If you use the **Developer Command Prompt for Visual Studio** window, all the necessary environment variables are set for you. For information on how to access this tool, see [Developer Command Prompt for Visual Studio](#).

The *csc.exe* executable file is usually located in the Microsoft.NET\Framework\<Version> folder under the *Windows* directory. Its location might vary depending on the exact configuration of a particular computer. If more than one version of .NET Framework is installed on your computer, you'll find multiple versions of this file. For more information about such installations, see [How to: determine which versions of the .NET Framework are installed](#).

C# Compiler Options for language feature rules

12/28/2021 • 7 minutes to read • [Edit Online](#)

The following options control how the compiler interprets language features. The new MSBuild syntax is shown in **Bold**. The older *csc.exe* syntax is shown in `code style`.

- **CheckForOverflowUnderflow** / `-checked`: Generate overflow checks.
- **AllowUnsafeBlocks** / `-unsafe`: Allow 'unsafe' code.
- **DefineConstants** / `-define`: Define conditional compilation symbol(s).
- **LangVersion** / `-langversion`: Specify language version such as `default` (latest major version), or `latest` (latest version, including minor versions).
- **Nullable** / `-nullable`: Enable nullable context, or nullable warnings.

CheckForOverflowUnderflow

The **CheckForOverflowUnderflow** option specifies whether an integer arithmetic statement that results in a value that is outside the range of the data type causes a run-time exception.

```
<CheckForOverflowUnderflow>true</CheckForOverflowUnderflow>
```

An integer arithmetic statement that is in the scope of a `checked` or `unchecked` keyword isn't subject to the effect of the **CheckForOverflowUnderflow** option. If an integer arithmetic statement that isn't in the scope of a `checked` or `unchecked` keyword results in a value outside the range of the data type, and **CheckForOverflowUnderflow** is `true`, that statement causes an exception at run time. If **CheckForOverflowUnderflow** is `false`, that statement doesn't cause an exception at run time. The default value for this option is `false`; overflow checking is disabled.

AllowUnsafeBlocks

The **AllowUnsafeBlocks** compiler option allows code that uses the `unsafe` keyword to compile. The default value for this option is `false`, meaning unsafe code is not allowed.

```
<AllowUnsafeBlocks>true</AllowUnsafeBlocks>
```

For more information about unsafe code, see [Unsafe Code and Pointers](#).

DefineConstants

The **DefineConstants** option defines symbols in all source code files of your program.

```
<DefineConstants>name;name2</DefineConstants>
```

This option specifies the names of one or more symbols that you want to define. The **DefineConstants** option has the same effect as the `#define` preprocessor directive except that the compiler option is in effect for all files in the project. A symbol remains defined in a source file until an `#undef` directive in the source file removes the definition. When you use the `-define` option, an `#undef` directive in one file has no effect on other source code files in the project. You can use symbols created by this option with `#if`, `#else`, `#elif`, and `#endif` to compile source

files conditionally. The C# compiler itself defines no symbols or macros that you can use in your source code; all symbol definitions must be user-defined.

NOTE

The C# `#define` directive does not allow a symbol to be given a value, as in languages such as C++. For example, `#define` cannot be used to create a macro or to define a constant. If you need to define a constant, use an `enum` variable. If you want to create a C++ style macro, consider alternatives such as generics. Since macros are notoriously error-prone, C# disallows their use but provides safer alternatives.

LangVersion

Causes the compiler to accept only syntax that is included in the chosen C# language specification.

```
<LangVersion>9.0</LangVersion>
```

The following values are valid:

VALUE	MEANING
<code>preview</code>	The compiler accepts all valid language syntax from the latest preview version.
<code>latest</code>	The compiler accepts syntax from the latest released version of the compiler (including minor version).
<code>latestMajor</code> (<code>default</code>)	The compiler accepts syntax from the latest released major version of the compiler.
<code>10.0</code>	The compiler accepts only syntax that is included in C# 10 or lower.
<code>9.0</code>	The compiler accepts only syntax that is included in C# 9 or lower.
<code>8.0</code>	The compiler accepts only syntax that is included in C# 8.0 or lower.
<code>7.3</code>	The compiler accepts only syntax that is included in C# 7.3 or lower.
<code>7.2</code>	The compiler accepts only syntax that is included in C# 7.2 or lower.
<code>7.1</code>	The compiler accepts only syntax that is included in C# 7.1 or lower.
<code>7</code>	The compiler accepts only syntax that is included in C# 7.0 or lower.
<code>6</code>	The compiler accepts only syntax that is included in C# 6.0 or lower.

VALUE	MEANING
5	The compiler accepts only syntax that is included in C# 5.0 or lower.
4	The compiler accepts only syntax that is included in C# 4.0 or lower.
3	The compiler accepts only syntax that is included in C# 3.0 or lower.
ISO-2 (or 2)	The compiler accepts only syntax that is included in ISO/IEC 23270:2006 C# (2.0).
ISO-1 (or 1)	The compiler accepts only syntax that is included in ISO/IEC 23270:2003 C# (1.0/1.2).

The default language version depends on the target framework for your application and the version of the SDK or Visual Studio installed. Those rules are defined in [C# language versioning](#).

Metadata referenced by your C# application isn't subject to the **LangVersion** compiler option.

Because each version of the C# compiler contains extensions to the language specification, **LangVersion** doesn't give you the equivalent functionality of an earlier version of the compiler.

Additionally, while C# version updates generally coincide with major .NET Framework releases, the new syntax and features aren't necessarily tied to that specific framework version. While the new features definitely require a new compiler update that is also released alongside the C# revision, each specific feature has its own minimum .NET API or common language runtime requirements that may allow it to run on downlevel frameworks by including NuGet packages or other libraries.

Regardless of which **LangVersion** setting you use, use the current version of the common language runtime to create your .exe or .dll. One exception is friend assemblies and **ModuleAssemblyName**, which work under -**langversion:ISO-1**.

For other ways to specify the C# language version, see [C# language versioning](#).

For information about how to set this compiler option programmatically, see [LanguageVersion](#).

C# language specification

VERSION	LINK	DESCRIPTION
C# 7.0 and later		Not currently available
C# 6.0	Link	C# Language Specification Version 6 - Unofficial Draft: .NET Foundation
C# 5.0	Download PDF	Standard ECMA-334 5th Edition
C# 3.0	Download DOC	C# Language Specification Version 3.0: Microsoft Corporation
C# 2.0	Download PDF	Standard ECMA-334 4th Edition
C# 1.2	Download DOC	C# Language Specification Version 1.2: Microsoft Corporation

VERSION	LINK	DESCRIPTION
C# 1.0	Download DOC	C# Language Specification Version 1.0: Microsoft Corporation

Minimum SDK version needed to support all language features

The following table lists the minimum versions of the SDK with the C# compiler that supports the corresponding language version:

C# VERSION	MINIMUM SDK VERSION
C# 10	Microsoft Visual Studio/Build Tools 2022, or .NET 6 SDK
C# 9.0	Microsoft Visual Studio/Build Tools 2019, version 16.8, or .NET 5 SDK
C# 8.0	Microsoft Visual Studio/Build Tools 2019, version 16.3, or .NET Core 3.0 SDK
C# 7.3	Microsoft Visual Studio/Build Tools 2017, version 15.7
C# 7.2	Microsoft Visual Studio/Build Tools 2017, version 15.5
C# 7.1	Microsoft Visual Studio/Build Tools 2017, version 15.3
C# 7.0	Microsoft Visual Studio/Build Tools 2017
C# 6	Microsoft Visual Studio/Build Tools 2015
C# 5	Microsoft Visual Studio/Build Tools 2012 or bundled .NET Framework 4.5 compiler
C# 4	Microsoft Visual Studio/Build Tools 2010 or bundled .NET Framework 4.0 compiler
C# 3	Microsoft Visual Studio/Build Tools 2008 or bundled .NET Framework 3.5 compiler
C# 2	Microsoft Visual Studio/Build Tools 2005 or bundled .NET Framework 2.0 compiler
C# 1.0/1.2	Microsoft Visual Studio/Build Tools .NET 2002 or bundled .NET Framework 1.0 compiler

Nullable

The **Nullable** option lets you specify the nullable context. The default value for this option is `disable`.

```
<Nullable>enable</Nullable>
```

The argument must be one of `enable`, `disable`, `warnings`, or `annotations`. The `enable` argument enables the nullable context. Specifying `disable` will disable the nullable context. When providing the `warnings` argument the nullable warning context is enabled. When specifying the `annotations` argument, the nullable annotation

context is enabled.

Flow analysis is used to infer the nullability of variables within executable code. The inferred nullability of a variable is independent of the variable's declared nullability. Method calls are analyzed even when they're conditionally omitted. For instance, [Debug.Assert](#) in release mode.

Invocation of methods annotated with the following attributes will also affect flow analysis:

- Simple pre-conditions: [AllowNullAttribute](#) and [DisallowNullAttribute](#)
- Simple post-conditions: [MaybeNullAttribute](#) and [NotNullAttribute](#)
- Conditional post-conditions: [MaybeNullWhenAttribute](#) and [NotNullWhenAttribute](#)
- [DoesNotReturnIfAttribute](#) (for example, `DoesNotReturnIf(false)` for [Debug.Assert](#)) and [DoesNotReturnAttribute](#)
- [NotNullIfNotNullAttribute](#)
- Member post-conditions: [MemberNotNullAttribute\(String\)](#) and [MemberNotNullAttribute\(String\[\]\)](#)

IMPORTANT

The global nullable context does not apply for generated code files. Regardless of this setting, the nullable context is *disabled* for any source file marked as generated. There are four ways a file is marked as generated:

1. In the .editorconfig, specify `generated_code = true` in a section that applies to that file.
2. Put `<auto-generated>` or `<auto-generated/>` in a comment at the top of the file. It can be on any line in that comment, but the comment block must be the first element in the file.
3. Start the file name with *TemporaryGeneratedFile_*
4. End the file name with *.designer.cs*, *.generated.cs*, *.g.cs*, or *.g.i.cs*.

Generators can opt-in using the `#nullable` preprocessor directive.

C# Compiler Options that control compiler output

12/28/2021 • 8 minutes to read • [Edit Online](#)

The following options control compiler output generation.

MSBUILD	CSC.EXE	DESCRIPTION
DocumentationFile	<code>-doc:</code>	Generate XML doc file from <code>///</code> comments.
OutputAssembly	<code>-out:</code>	Specify the output assembly file.
PlatformTarget	<code>-platform:</code>	Specify the target platform CPU.
ProduceReferenceAssembly	<code>-refout:</code>	Generate a reference assembly.
TargetType	<code>-target:</code>	Specify the type of the output assembly.

DocumentationFile

The **DocumentationFile** option allows you to place documentation comments in an XML file. To learn more about documenting your code, see [Recommended Tags for Documentation Comments](#). The value specifies the path to the output XML file. The XML file contains the comments in the source code files of the compilation.

```
<DocumentationFile>path/to/file.xml</DocumentationFile>
```

The source code file that contains Main or top-level statements is output first into the XML. You'll often want to use the generated .xml file with [IntelliSense](#). The .xml/ filename must be the same as the assembly name. The .xml/ file must be in the same directory as the assembly. When the assembly is referenced in a Visual Studio project, the .xml/ file is found as well. For more information about generating code comments, see [Supplying Code Comments](#). Unless you compile with `<TargetType:Module>`, `file` will contain `<assembly>` and `</assembly>` tags specifying the name of the file containing the assembly manifest for the output file. For examples, see [How to use the XML documentation features](#).

NOTE

The **DocumentationFile** option applies to all files in the project. To disable warnings related to documentation comments for a specific file or section of code, use [#pragma warning](#).

This option can be used in any .NET SDK-style project. For more information, see [DocumentationFile property](#).

OutputAssembly

The **OutputAssembly** option specifies the name of the output file. The output path specifies the folder where compiler output is placed.

```
<OutputAssembly>folder</OutputAssembly>
```

Specify the full name and extension of the file you want to create. If you don't specify the name of the output file, MSBuild uses the name of the project to specify the name of the output assembly. Old style projects use the following rules:

- An .exe will take its name from the source code file that contains the `Main` method or top-level statements.
- A .dll or .netmodule will take its name from the first source code file.

Any modules produced as part of a compilation become files associated with any assembly also produced in the compilation. Use [ildasm.exe](#) to view the assembly manifest to see the associated files.

The **OutputAssembly** compiler option is required in order for an exe to be the target of a [friend assembly](#).

PlatformTarget

Specifies which version of the CLR can run the assembly.

```
<PlatformTarget>anycpu</PlatformTarget>
```

- **anycpu** (default) compiles your assembly to run on any platform. Your application runs as a 64-bit process whenever possible and falls back to 32-bit when only that mode is available.
- **anycpu32bitpreferred** compiles your assembly to run on any platform. Your application runs in 32-bit mode on systems that support both 64-bit and 32-bit applications. You can specify this option only for projects that target .NET Framework 4.5 or later.
- **ARM** compiles your assembly to run on a computer that has an Advanced RISC Machine (ARM) processor.
- **ARM64** compiles your assembly to run by the 64-bit CLR on a computer that has an Advanced RISC Machine (ARM) processor that supports the A64 instruction set.
- **x64** compiles your assembly to be run by the 64-bit CLR on a computer that supports the AMD64 or EM64T instruction set.
- **x86** compiles your assembly to be run by the 32-bit, x86-compatible CLR.
- **Itanium** compiles your assembly to be run by the 64-bit CLR on a computer with an Itanium processor.

On a 64-bit Windows operating system:

- Assemblies compiled with **x86** execute on the 32-bit CLR running under WOW64.
- A DLL compiled with the **anycpu** executes on the same CLR as the process into which it's loaded.
- Executables that are compiled with the **anycpu** execute on the 64-bit CLR.
- Executables compiled with **anycpu32bitpreferred** execute on the 32-bit CLR.

The **anycpu32bitpreferred** setting is valid only for executable (.EXE) files, and it requires .NET Framework 4.5 or later. For more information about developing an application to run on a Windows 64-bit operating system, see [64-bit Applications](#).

You set the **PlatformTarget** option from **Build** properties page for your project in Visual Studio.

The behavior of **anycpu** has some additional nuances on .NET Core and .NET 5 and later releases. When you set **anycpu**, publish your app and execute it with either the x86 `dotnet.exe` or the x64 `dotnet.exe`. For self-contained apps, the `dotnet publish` step packages the executable for the configured RID.

ProduceReferenceAssembly

The **ProduceReferenceAssembly** option specifies a file path where the reference assembly should be output.

It translates to `metadataPeStream` in the Emit API. The `filepath` specifies the path for the reference assembly. It should generally match that of the primary assembly. The recommended convention (used by MSBuild) is to place the reference assembly in a "ref/" subfolder relative to the primary assembly.

```
<ProduceReferenceAssembly>filepath</ProduceReferenceAssembly>
```

Reference assemblies are a special type of assembly that contains only the minimum amount of metadata required to represent the library's public API surface. They include declarations for all members that are significant when referencing an assembly in build tools. Reference assemblies exclude all member implementations and declarations of private members. Those members have no observable impact on their API contract. For more information, see [Reference assemblies](#) in the .NET Guide.

The `ProduceReferenceAssembly` and `ProduceOnlyReferenceAssembly` options are mutually exclusive.

TargetType

The `TargetType` compiler option can be specified in one of the following forms:

- **library**: to create a code library. **library** is the default value.
- **exe**: to create an .exe file.
- **module** to create a module.
- **winexe** to create a Windows program.
- **winmdobj** to create an intermediate *.winmdobj* file.
- **appcontainerexe** to create an .exe file for Windows 8.x Store apps.

NOTE

For .NET Framework targets, unless you specify **module**, this option causes a .NET Framework assembly manifest to be placed in an output file. For more information, see [Assemblies in .NET](#) and [Common Attributes](#).

```
<TargetType>library</TargetType>
```

The compiler creates only one assembly manifest per compilation. Information about all files in a compilation is placed in the assembly manifest. When producing multiple output files at the command line, only one assembly manifest can be created and it must go into the first output file specified on the command line.

If you create an assembly, you can indicate that all or part of your code is CLS-compliant with the [CLSCompliantAttribute](#) attribute.

library

The **library** option causes the compiler to create a dynamic-link library (DLL) rather than an executable file (EXE). The DLL will be created with the *.dll* extension. Unless otherwise specified with the [OutputAssembly](#) option, the output file name takes the name of the first input file. When building a *.dll* file, a `Main` method isn't required.

exe

The **exe** option causes the compiler to create an executable (EXE), console application. The executable file will be created with the *.exe* extension. Use **winexe** to create a Windows program executable. Unless otherwise specified with the [OutputAssembly](#) option, the output file name takes the name of the input file that contains the entry point (`Main` method or top-level statements). One and only one entry point is required in the source code files that are compiled into an *.exe* file. The [StartupObject](#) compiler option lets you specify which class contains the `Main` method, in cases where your code has more than one class with a `Main` method.

module

This option causes the compiler to not generate an assembly manifest. By default, the output file created by compiling with this option will have an extension of *.netmodule*. A file that doesn't have an assembly manifest cannot be loaded by the .NET runtime. However, such a file can be incorporated into the assembly manifest of an assembly with [AddModules](#). If more than one module is created in a single compilation, [internal](#) types in one module will be available to other modules in the compilation. When code in one module references `internal` types in another module, then both modules must be incorporated into an assembly manifest, with [AddModules](#). Creating a module isn't supported in the Visual Studio development environment.

winexe

The **winexe** option causes the compiler to create an executable (EXE), Windows program. The executable file will be created with the .exe extension. A Windows program is one that provides a user interface from either the .NET library or with the Windows APIs. Use **exe** to create a console application. Unless otherwise specified with the [OutputAssembly](#) option, the output file name takes the name of the input file that contains the `Main` method. One and only one `Main` method is required in the source code files that are compiled into an .exe file. The [StartupObject](#) option lets you specify which class contains the `Main` method, in cases where your code has more than one class with a `Main` method.

winmdobj

If you use the **winmdobj** option, the compiler creates an intermediate *.winmdobj* file that you can convert to a Windows Runtime binary (*.winmd*) file. The *.winmd* file can then be consumed by JavaScript and C++ programs, in addition to managed language programs.

The **winmdobj** setting signals to the compiler that an intermediate module is required. The *.winmdobj* file can then be fed through the [WinMDExp](#) export tool to produce a Windows metadata (*.winmd*) file. The *.winmd* file contains both the code from the original library and the WinMD metadata that is used by JavaScript or C++ and by the Windows Runtime. The output of a file that's compiled by using the **winmdobj** compiler option is used only as input for the WinMDExp export tool. The *.winmdobj* file itself isn't referenced directly. Unless you use the [OutputAssembly](#) option, the output file name takes the name of the first input file. A `Main` method isn't required.

appcontainerexe

If you use the **appcontainerexe** compiler option, the compiler creates a Windows executable (.exe) file that must be run in an app container. This option is equivalent to `-target:winexe` but is designed for Windows 8.x Store apps.

To require the app to run in an app container, this option sets a bit in the [Portable Executable](#) (PE) file. When that bit is set, an error occurs if the `CreateProcess` method tries to launch the executable file outside an app container. Unless you use the [OutputAssembly](#) option, the output file name takes the name of the input file that contains the `Main` method.

C# Compiler Options that specify inputs

12/28/2021 • 5 minutes to read • [Edit Online](#)

The following options control compiler inputs. The new MSBuild syntax is shown in **Bold**. The older *csc.exe* syntax is shown in `code style`.

- **References** / `-reference` or `-references`: Reference metadata from the specified assembly file or files.
- **AddModules** / `-addmodule`: Add a module (created with `target:module` to this assembly.)
- **EmbedInteropTypes** / `-link`: Embed metadata from the specified interop assembly files.

References

The **References** option causes the compiler to import [public](#) type information in the specified file into the current project, enabling you to reference metadata from the specified assembly files.

```
<Reference Include="filename" />
```

`filename` is the name of a file that contains an assembly manifest. To import more than one file, include a separate **Reference** element for each file. You can define an alias as a child element of the **Reference** element:

```
<Reference Include="filename.dll">  
  <Aliases>LS</Aliases>  
</Reference>
```

In the previous example, `LS` is the valid C# identifier that represents a root namespace that will contain all namespaces in the assembly *filename.dll*. The files you import must contain a manifest. Use [AdditionalLibPaths](#) to specify the directory in which one or more of your assembly references is located. The [AdditionalLibPaths](#) topic also discusses the directories in which the compiler searches for assemblies. In order for the compiler to recognize a type in an assembly, and not in a module, it needs to be forced to resolve the type, which you can do by defining an instance of the type. There are other ways to resolve type names in an assembly for the compiler: for example, if you inherit from a type in an assembly, the type name will then be recognized by the compiler. Sometimes it is necessary to reference two different versions of the same component from within one assembly. To do this, use the **Aliases** element on the **References** element for each file to distinguish between the two files. This alias will be used as a qualifier for the component name, and will resolve to the component in one of the files.

NOTE

In Visual Studio, use the **Add Reference** command. For more information, see [How to: Add or Remove References By Using the Reference Manager](#).

AddModules

This option adds a module that was created with the `<TargetType>module</TargetType>` switch to the current compilation:


```
<AddModule Include=file1 />
<AddModule Include=file2 />
```

Where `file`, `file2` are output files that contain metadata. The file can't contain an assembly manifest. To import more than one file, separate file names with either a comma or a semicolon. All modules added with **AddModules** must be in the same directory as the output file at run time. That is, you can specify a module in any directory at compile time but the module must be in the application directory at run time. If the module isn't in the application directory at run time, you'll get a [TypeLoadException](#). `file` can't contain an assembly. For example, if the output file was created with **TargetType** option of **module**, its metadata can be imported with **AddModules**.

If the output file was created with a **TargetType** option other than **module**, its metadata cannot be imported with **AddModules** but can be imported with the **References** option.

EmbedInteropTypes

Causes the compiler to make COM type information in the specified assemblies available to the project that you are currently compiling.

```
<References>
  <EmbedInteropTypes>file1;file2;file3</EmbedInteropTypes>
</References>
```

Where `file1;file2;file3` is a semicolon-delimited list of assembly file names. If the file name contains a space, enclose the name in quotation marks. The **EmbedInteropTypes** option enables you to deploy an application that has embedded type information. The application can then use types in a runtime assembly that implement the embedded type information without requiring a reference to the runtime assembly. If various versions of the runtime assembly are published, the application that contains the embedded type information can work with the various versions without having to be recompiled. For an example, see [Walkthrough: Embedding Types from Managed Assemblies](#).

Using the **EmbedInteropTypes** option is especially useful when you're working with COM interop. You can embed COM types so that your application no longer requires a primary interop assembly (PIA) on the target computer. The **EmbedInteropTypes** option instructs the compiler to embed the COM type information from the referenced interop assembly into the resulting compiled code. The COM type is identified by the CLSID (GUID) value. As a result, your application can run on a target computer that has installed the same COM types with the same CLSID values. Applications that automate Microsoft Office are a good example. Because applications like Office usually keep the same CLSID value across different versions, your application can use the referenced COM types as long as .NET Framework 4 or later is installed on the target computer and your application uses methods, properties, or events that are included in the referenced COM types. The **EmbedInteropTypes** option embeds only interfaces, structures, and delegates. Embedding COM classes isn't supported.

NOTE

When you create an instance of an embedded COM type in your code, you must create the instance by using the appropriate interface. Attempting to create an instance of an embedded COM type by using the `CoClass` causes an error.

Like the **References** compiler option, the **EmbedInteropTypes** compiler option uses the `Csc.rsp` response file, which references frequently used .NET assemblies. Use the **NoConfig** compiler option if you don't want the compiler to use the `Csc.rsp` file.

```
// The following code causes an error if ISampleInterface is an embedded interop type.  
ISampleInterface<SampleType> sample;
```

Types that have a generic parameter whose type is embedded from an interop assembly cannot be used if that type is from an external assembly. This restriction doesn't apply to interfaces. For example, consider the [Range](#) interface that is defined in the [Microsoft.Office.Interop.Excel](#) assembly. If a library embeds interop types from the [Microsoft.Office.Interop.Excel](#) assembly and exposes a method that returns a generic type that has a parameter whose type is the [Range](#) interface, that method must return a generic interface, as shown in the following code example.

```
using System;  
using System.Collections.Generic;  
using System.Linq;  
using System.Text;  
using Microsoft.Office.Interop.Excel;  
  
public class Utility  
{  
    // The following code causes an error when called by a client assembly.  
    public List<Range> GetRange1()  
    {  
        return null;  
    }  
  
    // The following code is valid for calls from a client assembly.  
    public IList<Range> GetRange2()  
    {  
        return null;  
    }  
}
```

In the following example, client code can call the method that returns the [IList](#) generic interface without error.

```
public class Client  
{  
    public void Main()  
    {  
        Utility util = new Utility();  
  
        // The following code causes an error.  
        List<Range> rangeList1 = util.GetRange1();  
  
        // The following code is valid.  
        List<Range> rangeList2 = (List<Range>)util.GetRange2();  
    }  
}
```

C# Compiler Options to report errors and warnings

12/28/2021 • 3 minutes to read • [Edit Online](#)

The following options control how the compiler reports errors and warnings. The new MSBuild syntax is shown in **Bold**. The older *csc.exe* syntax is shown in `code style`.

- **WarningLevel** / `-warn` : Set warning level.
- **TreatWarningsAsErrors** / `-warnaserror` : Treat all warnings as errors
- **WarningsAsErrors** / `-warnaserror` : Treat one or more warnings as errors
- **WarningsNotAsErrors** / `-warnnotaserror` : Treat one or more warnings not as errors
- **DisabledWarnings** / `-nowarn` : Set a list of disabled warnings.
- **CodeAnalysisRuleSet** / `-ruleset` : Specify a ruleset file that disables specific diagnostics.
- **ErrorLog** / `-errorlog` : Specify a file to log all compiler and analyzer diagnostics.
- **ReportAnalyzer** / `-reportanalyzer` : Report additional analyzer information, such as execution time.

WarningLevel

The **WarningLevel** option specifies the warning level for the compiler to display.

```
<WarningLevel>3</WarningLevel>
```

The element value is the warning level you want displayed for the compilation: Lower numbers show only high severity warnings. Higher numbers show more warnings. The value must be zero or a positive integer:

WARNING LEVEL	MEANING
0	Turns off emission of all warning messages.
1	Displays severe warning messages.
2	Displays level 1 warnings plus certain, less-severe warnings, such as warnings about hiding class members.
3	Displays level 2 warnings plus certain, less-severe warnings, such as warnings about expressions that always evaluate to <code>true</code> or <code>false</code> .
4 (the default)	Displays all level 3 warnings plus informational warnings.
5	Displays level 4 warnings plus additional warnings from the compiler shipped with C# 9.0.
Greater than 5	Any value greater than 5 will be treated as 5. To make sure you always have all warnings if the compiler is updated with new warning levels, put an arbitrary large value (for example, <code>9999</code>).

To get information about an error or warning, you can look up the error code in the Help Index. For other ways to get information about an error or warning, see [C# Compiler Errors](#). Use **TreatWarningsAsErrors** to treat all

warnings as errors. Use [DisabledWarnings](#) to disable certain warnings.

TreatWarningsAsErrors

The **TreatWarningsAsErrors** option treats all warnings as errors. You can also use the **TreatWarningsAsErrors** to set only some warnings as errors. If you turn on **TreatWarningsAsErrors**, you can use **TreatWarningsAsErrors** to list warnings that shouldn't be treated as errors.

```
<TreatWarningsAsErrors>true</TreatWarningsAsErrors>
```

All warning messages are instead reported as errors. The build process halts (no output files are built). By default, **TreatWarningsAsErrors** isn't in effect, which means warnings don't prevent the generation of an output file. Optionally, if you want only a few specific warnings to be treated as errors, you may specify a comma-separated list of warning numbers to treat as errors. The set of all nullability warnings can be specified with the [Nullable](#) shorthand. Use [WarningLevel](#) to specify the level of warnings that you want the compiler to display. Use [DisabledWarnings](#) to disable certain warnings.

WarningsAsErrors and WarningsNotAsErrors

The **WarningsAsErrors** and **WarningsNotAsErrors** options override the **TreatWarningsAsErrors** option for a list of warnings.

Enable warnings 0219 and 0168 as errors:

```
<WarningsAsErrors>0219,0168</WarningsAsErrors>
```

Disable the same warnings as errors:

```
<WarningsNotAsErrors>0219,0168</WarningsNotAsErrors>
```

You use **WarningsAsErrors** to configure a set of warnings as errors. Use **WarningsNotAsErrors** to configure a set of warnings that should not be errors when you've set all warnings as errors.

DisabledWarnings

The **DisabledWarnings** option lets you suppress the compiler from displaying one or more warnings. Separate multiple warning numbers with a comma.

```
<DisabledWarnings>number1, number2</DisabledWarnings>
```

`number1`, `number2` Warning number(s) that you want the compiler to suppress. You specify the numeric part of the warning identifier. For example, if you want to suppress *CS0028*, you could specify

`<DisabledWarnings>28</DisabledWarnings>`. The compiler silently ignores warning numbers passed to **DisabledWarnings** that were valid in previous releases, but that have been removed. For example, *CS0679* was valid in the compiler in Visual Studio .NET 2002 but was removed later.

The following warnings cannot be suppressed by the **DisabledWarnings** option:

- Compiler Warning (level 1) CS2002
- Compiler Warning (level 1) CS2023
- Compiler Warning (level 1) CS2029

CodeAnalysisRuleSet

Specify a ruleset file that configures specific diagnostics.

```
<CodeAnalysisRuleSet>MyConfiguration.ruleset</CodeAnalysisRuleSet>
```

Where `MyConfiguration.ruleset` is the path to the ruleset file. For more information on using rule sets, see the article in the [Visual Studio documentation on Rule sets](#).

ErrorLog

Specify a file to log all compiler and analyzer diagnostics.

```
<ErrorLog>compiler-diagnostics.sarif</ErrorLog>
```

The **ErrorLog** option causes the compiler to output a [Static Analysis Results Interchange Format \(SARIF\) log](#). SARIF logs are typically read by tools that analyze the results from compiler and analyzer diagnostics.

ReportAnalyzer

Report additional analyzer information, such as execution time.

```
<ReportAnalyzer>true</ReportAnalyzer>
```

The **ReportAnalyzer** option causes the compiler to emit extra MSBuild log information that details the performance characteristics of analyzers in the build. It's typically used by analyzer authors as part of validating the analyzer.

C# Compiler Options that control code generation

12/28/2021 • 4 minutes to read • [Edit Online](#)

The following options control code generation by the compiler. The new MSBuild syntax is shown in **Bold**. The older *csc.exe* syntax is shown in `code style`.

- **DebugType** / `-debug`: Emit (or don't emit) debugging information.
- **Optimize** / `-optimize`: Enable optimizations.
- **Deterministic** / `-deterministic`: Produce byte-for-byte equivalent output from the same input source.
- **ProduceOnlyReferenceAssembly** / `-refonly`: Produce a reference assembly, instead of a full assembly, as the primary output.

DebugType

The **DebugType** option causes the compiler to generate debugging information and place it in the output file or files. Debugging information is added by default for the *Debug* build configuration. It is off by default for the *Release* build configuration.

```
<DebugType>pdbonly</DebugType>
```

For all compiler versions starting with C# 6.0, there is no difference between *pdbonly* and *full*. Choose *pdbonly*. To change the location of the *.pdb* file, see [PdbFile](#).

The following values are valid:

VALUE	MEANING
<code>full</code>	Emit debugging information to <i>.pdb</i> file using default format for the current platform: Windows: A Windows pdb file. Linux/macOS: A Portable PDB file.
<code>pdbonly</code>	Same as <code>full</code> . See the note below for more information.
<code>portable</code>	Emit debugging information to <i>.pdb</i> file using cross-platform Portable PDB format.
<code>embedded</code>	Emit debugging information into the <i>.dll/.exe</i> itself (<i>.pdb</i> file is not produced) using Portable PDB format.

IMPORTANT

The following information applies only to compilers older than C# 6.0. The value of this element can be either `full` or `pdbonly`. The *full* argument, which is in effect if you don't specify *pdbonly*, enables attaching a debugger to the running program. Specifying *pdbonly* allows source code debugging when the program is started in the debugger but will only display assembler when the running program is attached to the debugger. Use this option to create debug builds. If you use *Full*, be aware that there's some impact on the speed and size of JIT optimized code and a small impact on code quality with *full*. We recommend *pdbonly* or no PDB for generating release code. One difference between *pdbonly* and *full* is that with *full* the compiler emits a [DebuggableAttribute](#), which is used to tell the JIT compiler that debug information is available. Therefore, you will get an error if your code contains the [DebuggableAttribute](#) set to false if you use *full*. For more information on how to configure the debug performance of an application, see [Making an Image Easier to Debug](#).

Optimize

The **Optimize** option enables or disables optimizations performed by the compiler to make your output file smaller, faster, and more efficient. The *Optimize* option is enabled by default for a *Release* build configuration. It is off by default for a *Debug* build configuration.

```
<Optimize>true</Optimize>
```

You set the **Optimize** option from **Build** properties page for your project in Visual Studio.

Optimize also tells the common language runtime to optimize code at run time. By default, optimizations are disabled. Specify **Optimize+** to enable optimizations. When building a module to be used by an assembly, use the same **Optimize** settings as used by the assembly. It's possible to combine the **Optimize** and [Debug](#) options.

Deterministic

Causes the compiler to produce an assembly whose byte-for-byte output is identical across compilations for identical inputs.

```
<Deterministic>true</Deterministic>
```

By default, compiler output from a given set of inputs is unique, since the compiler adds a timestamp and an MVID that is generated from random numbers. You use the `<Deterministic>` option to produce a *deterministic assembly*, one whose binary content is identical across compilations as long as the input remains the same. In such a build, the timestamp and MVID fields will be replaced with values derived from a hash of all the compilation inputs. The compiler considers the following inputs that affect determinism:

- The sequence of command-line parameters.
- The contents of the compiler's .rsp response file.
- The precise version of the compiler used, and its referenced assemblies.
- The current directory path.
- The binary contents of all files explicitly passed to the compiler either directly or indirectly, including:
 - Source files
 - Referenced assemblies
 - Referenced modules
 - Resources
 - The strong name key file
 - @ response files

- Analyzers
- Rulesets
- Other files that may be used by analyzers
- The current culture (for the language in which diagnostics and exception messages are produced).
- The default encoding (or the current code page) if the encoding isn't specified.
- The existence, non-existence, and contents of files on the compiler's search paths (specified, for example, by `-lib` or `-recurse`).
- The Common Language Runtime (CLR) platform on which the compiler is run.
- The value of `%LIBPATH%`, which can affect analyzer dependency loading.

Deterministic compilation can be used for establishing whether a binary is compiled from a trusted source. Deterministic output can be useful when the source is publicly available. It can also determine whether build steps that are dependent on changes to binary used in the build process.

ProduceOnlyReferenceAssembly

The **ProduceOnlyReferenceAssembly** option indicates that a reference assembly should be output instead of an implementation assembly, as the primary output. The **ProduceOnlyReferenceAssembly** parameter silently disables outputting PDBs, as reference assemblies cannot be executed.

```
<ProduceOnlyReferenceAssembly>true</ProduceOnlyReferenceAssembly>
```

Reference assemblies are a special type of assembly. Reference assemblies contain only the minimum amount of metadata required to represent the library's public API surface. They include declarations for all members that are significant when referencing an assembly in build tools, but exclude all member implementations and declarations of private members that have no observable impact on their API contract. For more information, see [Reference assemblies](#).

The **ProduceOnlyReferenceAssembly** and **ProduceReferenceAssembly** options are mutually exclusive.

C# compiler options for security

12/28/2021 • 4 minutes to read • [Edit Online](#)

The following options control compiler security options. The new MSBuild syntax is shown in **Bold**. The older *csc.exe* syntax is shown in `code style`.

- **PublicSign** / `-publicsign` : Publicly sign the assembly.
- **DelaySign** / `-delaysign` : Delay-sign the assembly using only the public portion of the strong name key.
- **KeyFile** / `-keyfile` : Specify a strong name key file.
- **KeyContainer** / `-keycontainer` : Specify a strong name key container.
- **HighEntropyVA** / `-highentropyva` : Enable high-entropy Address Space Layout Randomization (ASLR)

PublicSign

This option causes the compiler to apply a public key but doesn't actually sign the assembly. The **PublicSign** option also sets a bit in the assembly that tells the runtime that the file is signed.

```
<PublicSign>true</PublicSign>
```

The **PublicSign** option requires the use of the [KeyFile](#) or [KeyContainer](#) option. The [keyFile](#) and [KeyContainer](#) options specify the public key. The **PublicSign** and **DelaySign** options are mutually exclusive. Sometimes called "fake sign" or "OSS sign", public signing includes the public key in an output assembly and sets the "signed" flag. Public signing doesn't actually sign the assembly with a private key. Developers use public sign for open-source projects. People build assemblies that are compatible with the released "fully signed" assemblies when they don't have access to the private key used to sign the assemblies. Since few consumers actually need to check if the assembly is fully signed, these publicly built assemblies are useable in almost every scenario where the fully signed one would be used.

DelaySign

This option causes the compiler to reserve space in the output file so that a digital signature can be added later.

```
<DelaySign>true</DelaySign>
```

Use **DelaySign**- if you want a fully signed assembly. Use **DelaySign** if you only want to place the public key in the assembly. The **DelaySign** option has no effect unless used with [KeyFile](#) or [KeyContainer](#). The [KeyContainer](#) and [PublicSign](#) options are mutually exclusive. When you request a fully signed assembly, the compiler hashes the file that contains the manifest (assembly metadata) and signs that hash with the private key. That operation creates a digital signature that is stored in the file that contains the manifest. When an assembly is delay signed, the compiler doesn't compute and store the signature. Instead, the compiler but reserves space in the file so the signature can be added later.

Using **DelaySign** allows a tester to put the assembly in the global cache. After testing, you can fully sign the assembly by placing the private key in the assembly using the [Assembly Linker](#) utility. For more information, see [Creating and Using Strong-Named Assemblies](#) and [Delay Signing an Assembly](#).

KeyFile

Specifies the filename containing the cryptographic key.

```
<KeyFile>filename</KeyFile>
```

`file` is the name of the file containing the strong name key. When this option is used, the compiler inserts the public key from the specified file into the assembly manifest and then signs the final assembly with the private key. To generate a key file, type `sn -k file` at the command line. If you compile with **-target:module**, the name of the key file is held in the module and incorporated into the assembly created when you compile an assembly with **AddModules**. You can also pass your encryption information to the compiler with **KeyContainer**. Use **DelaySign** if you want a partially signed assembly. In case both **KeyFile** and **KeyContainer** are specified in the same compilation, the compiler will first try the key container. If that succeeds, then the assembly is signed with the information in the key container. If the compiler doesn't find the key container, it will try the file specified with **KeyFile**. If that succeeds, the assembly is signed with the information in the key file and the key information will be installed in the key container. On the next compilation, the key container will be valid. A key file might contain only the public key. For more information, see [Creating and Using Strong-Named Assemblies](#) and [Delay Signing an Assembly](#).

KeyContainer

Specifies the name of the cryptographic key container.

```
<KeyContainer>container</KeyContainer>
```

`container` is the name of the strong name key container. When the **KeyContainer** option is used, the compiler creates a sharable component. The compiler inserts a public key from the specified container into the assembly manifest and signs the final assembly with the private key. To generate a key file, type `sn -k file` at the command line. `sn -i` installs the key pair into a container. This option isn't supported when the compiler runs on CoreCLR. To sign an assembly when building on CoreCLR, use the **KeyFile** option. If you compile with **TargetType**, the name of the key file is held in the module and incorporated into the assembly when you compile this module into an assembly with **AddModules**. You can also specify this option as a custom attribute (**System.Reflection.AssemblyKeyNameAttribute**) in the source code for any Microsoft intermediate language (MSIL) module. You can also pass your encryption information to the compiler with **KeyFile**. Use **DelaySign** to add the public key to the assembly manifest but signing the assembly until it has been tested. For more information, see [Creating and Using Strong-Named Assemblies](#) and [Delay Signing an Assembly](#).

HighEntropyVA

The **HighEntropyVA** compiler option tells the Windows kernel whether a particular executable supports high entropy Address Space Layout Randomization (ASLR).

```
<HighEntropyVA>true</HighEntropyVA>
```

This option specifies that a 64-bit executable or an executable that is marked by the **PlatformTarget** compiler option supports a high entropy virtual address space. The option is disabled by default. Use **HighEntropyVA** to enable it.

The **HighEntropyVA** option enables compatible versions of the Windows kernel to use higher degrees of entropy when randomizing the address space layout of a process as part of ASLR. Using higher degrees of entropy means a larger number of addresses can be allocated to memory regions such as stacks and heaps. As a result, it's more difficult to guess the location of a particular memory region. The **HighEntropyVA** compiler option requires the target executable and any modules that it depends on can handle pointer values larger than

4 gigabytes (GB) when they're running as a 64-bit process.

C# Compiler Options that specify resources

12/28/2021 • 5 minutes to read • [Edit Online](#)

The following options control how the C# compiler creates or imports Win32 resources. The new MSBuild syntax is shown in **Bold**. The older *csc.exe* syntax is shown in `code style`.

- **Win32Resource** / `-win32res`: Specify a Win32 resource file (.res).
- **Win32Icon** / `-win32icon`: Reference metadata from the specified assembly file or files.
- **Win32Manifest** / `-win32manifest`: Specify a Win32 manifest file (.xml).
- **NoWin32Manifest** / `-nowin32manifest`: Don't include the default Win32 manifest.
- **Resources** / `-resource`: Embed the specified resource (Short form: /res).
- **LinkResources** / `-linkresources`: Link the specified resource to this assembly.

Win32Resource

The **Win32Resource** option inserts a Win32 resource in the output file.

```
<Win32Resource>filename</Win32Resource>
```

`filename` is the resource file that you want to add to your output file. A Win32 resource can contain version or bitmap (icon) information that would help identify your application in the File Explorer. If you don't specify this option, the compiler will generate version information based on the assembly version.

Win32Icon

The **Win32Icon** option inserts an .ico file in the output file, which gives the output file the desired appearance in the File Explorer.

```
<Win32Icon>filename</Win32Icon>
```

`filename` is the .ico file that you want to add to your output file. An .ico file can be created with the [Resource Compiler](#). The Resource Compiler is invoked when you compile a Visual C++ program; an .ico file is created from the .rc file.

Win32Manifest

Use the **Win32Manifest** option to specify a user-defined Win32 application manifest file to be embedded into a project's portable executable (PE) file.

```
<Win32Manifest>filename</Win32Manifest>
```

`filename` is the name and location of the custom manifest file. By default, the C# compiler embeds an application manifest that specifies a requested execution level of "asInvoker". It creates the manifest in the same folder in which the executable is built. If you want to supply a custom manifest, for example to specify a requested execution level of "highestAvailable" or "requireAdministrator," use this option to specify the name of the file.

NOTE

This option and the **Win32Resources** option are mutually exclusive. If you try to use both options in the same command line you will get a build error.

An application that has no application manifest that specifies a requested execution level will be subject to file and registry virtualization under the User Account Control feature in Windows. For more information, see [User Account Control](#).

Your application will be subject to virtualization if either of these conditions is true:

- You use the **NoWin32Manifest** option and you don't provide a manifest in a later build step or as part of a Windows Resource (.res) file by using the **Win32Resource** option.
- You provide a custom manifest that doesn't specify a requested execution level.

Visual Studio creates a default *.manifest* file and stores it in the debug and release directories alongside the executable file. You can add a custom manifest by creating one in any text editor and then adding the file to the project. Or, you can right-click the **Project** icon in **Solution Explorer**, select **Add New Item**, and then select **Application Manifest File**. After you've added your new or existing manifest file, it will appear in the **Manifest** drop down list. For more information, see [Application Page, Project Designer \(C#\)](#).

You can provide the application manifest as a custom post-build step or as part of a Win32 resource file by using the **NoWin32Manifest** option. Use that same option if you want your application to be subject to file or registry virtualization on Windows Vista.

NoWin32Manifest

Use the **NoWin32Manifest** option to instruct the compiler not to embed any application manifest into the executable file.

```
<NoWin32Manifest />
```

When this option is used, the application will be subject to virtualization on Windows Vista unless you provide an application manifest in a Win32 Resource file or during a later build step.

In Visual Studio, set this option in the **Application Property** page by selecting the **Create Application Without a Manifest** option in the **Manifest** drop down list. For more information, see [Application Page, Project Designer \(C#\)](#).

Resources

Embeds the specified resource into the output file.

```
<Resources Include=filename>  
  <LogicalName>identifier</LogicalName>  
  <Access>accessibility-modifier</Access>  
</Resources>
```

`filename` is the .NET resource file that you want to embed in the output file. `identifier` (optional) is the logical name for the resource; the name that is used to load the resource. The default is the name of the file.

`accessibility-modifier` (optional) is the accessibility of the resource: public or private. The default is public. By default, resources are public in the assembly when they're created by using the C# compiler. To make the resources private, specify `private` as the accessibility modifier. No other accessibility other than `public` or

`private` is allowed. If `filename` is a .NET resource file created, for example, by [Resgen.exe](#) or in the development environment, it can be accessed with members in the [System.Resources](#) namespace. For more information, see [System.Resources.ResourceManager](#). For all other resources, use the `GetManifestResource` methods in the [Assembly](#) class to access the resource at run time. The order of the resources in the output file is determined from the order specified in the project file.

LinkResources

Creates a link to a .NET resource in the output file. The resource file isn't added to the output file.

LinkResources differs from the **Resource** option, which does embed a resource file in the output file.

```
<LinkResources Include=filename>
  <LogicalName>identifier</LogicalName>
  <Access>accessibility-modifier</Access>
</LinkResources>
```

`filename` is the .NET resource file to which you want to link from the assembly. `identifier` (optional) is the logical name for the resource; the name that is used to load the resource. The default is the name of the file. `accessibility-modifier` (optional) is the accessibility of the resource: public or private. The default is public. By default, linked resources are public in the assembly when they're created with the C# compiler. To make the resources private, specify `private` as the accessibility modifier. No other modifier other than `public` or `private` is allowed. If `filename` is a .NET resource file created, for example, by [Resgen.exe](#) or in the development environment, it can be accessed with members in the [System.Resources](#) namespace. For more information, see [System.Resources.ResourceManager](#). For all other resources, use the `GetManifestResource` methods in the [Assembly](#) class to access the resource at run time. The file specified in `filename` can be any format. For example, you may want to make a native DLL part of the assembly, so that it can be installed into the global assembly cache and accessed from managed code in the assembly. You can do the same thing in the Assembly Linker. For more information, see [Al.exe \(Assembly Linker\)](#) and [Working with Assemblies and the Global Assembly Cache](#).

Miscellaneous C# Compiler Options

12/28/2021 • 2 minutes to read • [Edit Online](#)

The following options control miscellaneous compiler behavior. The new MSBuild syntax is shown in **Bold**. The older *csc.exe* syntax is shown in `code style`.

- **ResponseFiles** / `-@`: Read response file for more options.
- **NoLogo** / `-nologo`: Suppress compiler copyright message.
- **NoConfig** / `-noconfig`: Don't auto include *CSC.RSP* file.

ResponseFiles

The **ResponseFiles** option lets you specify a file that contains compiler options and source code files to compile.

```
<ResponseFiles>response_file</ResponseFiles>
```

The `response_file` specifies the file that lists compiler options or source code files to compile. The compiler options and source code files will be processed by the compiler as if they had been specified on the command line. To specify more than one response file in a compilation, specify multiple response file options. In a response file, multiple compiler options and source code files can appear on one line. A single compiler option specification must appear on one line (can't span multiple lines). Response files can have comments that begin with the # symbol. Specifying compiler options from within a response file is just like issuing those commands on the command line. The compiler processes the command options as they're read. Command-line arguments can override previously listed options in response files. Conversely, options in a response file will override options listed previously on the command line or in other response files. C# provides the *csc.rsp* file, which is located in the same directory as the *csc.exe* file. For more information about the response file format, see [NoConfig](#). This compiler option cannot be set in the Visual Studio development environment, nor can it be changed programmatically. The following are a few lines from a sample response file:

```
# build the first output file
-target:exe -out:MyExe.exe source1.cs source2.cs
```

NoLogo

The **NoLogo** option suppresses display of the sign-on banner when the compiler starts up and display of informational messages during compiling.

```
<NoLogo>true</NoLogo>
```

NoConfig

The **NoConfig** option tells the compiler not to compile with the *csc.rsp* file.

```
<NoConfig>true</NoConfig>
```

The *csc.rsp* file references all the assemblies shipped with .NET Framework. The actual references that the Visual

Studio .NET development environment includes depend on the project type. You can modify the *csc.rsp* file and specify additional compiler options that should be included in every compilation. If you don't want the compiler to look for and use the settings in the *csc.rsp* file, specify **NoConfig**. This compiler option is unavailable in Visual Studio and cannot be changed programmatically.

Advanced C# compiler options

12/28/2021 • 10 minutes to read • [Edit Online](#)

The following options support advanced scenarios. The new MSBuild syntax is shown in **Bold**. The older `csc.exe` syntax is shown in `code style`.

- **MainEntryPoint**, **StartupObject** / `-main` : Specify the type that contains the entry point.
- **PdbFile** / `-pdb` : Specify debug information file name.
- **PathMap** / `-pathmap` : Specify a mapping for source path names output by the compiler.
- **ApplicationConfiguration** / `-appconfig` : Specify an application configuration file containing assembly binding settings.
- **AdditionalLibPaths** / `-lib` : Specify additional directories to search in for references.
- **GenerateFullPaths** / `-fullpath` : Compiler generates fully qualified paths.
- **PreferredUILang** / `-preferredUILang` : Specify the preferred output language name.
- **BaseAddress** / `-baseaddress` : Specify the base address for the library to be built.
- **ChecksumAlgorithm** / `-checksumAlgorithm` : Specify algorithm for calculating source file checksum stored in PDB.
- **CodePage** / `-codepage` : Specify the codepage to use when opening source files.
- **Utf8Output** / `-utf8output` : Output compiler messages in UTF-8 encoding.
- **FileAlignment** / `-filealign` : Specify the alignment used for output file sections.
- **ErrorEndLocation** / `-errorEndLocation` : Output line and column of the end location of each error.
- **NoStandardLib** / `-nostdlib` : Don't reference standard library *mscorlib.dll*.
- **SubsystemVersion** / `-subsystemversion` : Specify subsystem version of this assembly.
- **ModuleAssemblyName** / `-moduleAssemblyName` : Name of the assembly that this module will be a part of.

MainEntryPoint or StartupObject

This option specifies the class that contains the entry point to the program, if more than one class contains a `Main` method.

```
<StartupObject>MyNamespace.Program</StartupObject>
```

or

```
<MainEntryPoint>MyNamespace.Program</MainEntryPoint>
```

Where `Program` is the type that contains the `Main` method. The provided class name must be fully qualified; it must include the full namespace containing the class, followed by the class name. For example, when the `Main` method is located inside the `Program` class in the `MyApplication.Core` namespace, the compiler option has to be `-main:MyApplication.Core.Program`. If your compilation includes more than one type with a `Main` method, you can specify which type contains the `Main` method.

NOTE

This option can't be used for a project that includes [top-level statements](#), even if that project contains one or more `Main` methods.

PdbFile

The **PdbFile** compiler option specifies the name and location of the debug symbols file. The `filename` value specifies the name and location of the debug symbols file.

```
<PdbFile>filename</PdbFile>
```

When you specify [DebugType](#), the compiler will create a *.pdb* file in the same directory where the compiler will create the output file (.exe or .dll). The *.pdb* file has the same base file name as the name of the output file.

PdbFile allows you to specify a non-default file name and location for the *.pdb* file. This compiler option cannot be set in the Visual Studio development environment, nor can it be changed programmatically.

PathMap

The **PathMap** compiler option specifies how to map physical paths to source path names output by the compiler. This option maps each physical path on the machine where the compiler runs to a corresponding path that should be written in the output files. In the following example, `path1` is the full path to the source files in the current environment, and `sourcePath1` is the source path substituted for `path1` in any output files. To specify multiple mapped source paths, separate each with a semicolon.

```
<PathMap>path1=sourcePath1;path2=sourcePath2</PathMap>
```

The compiler writes the source path into its output for the following reasons:

1. The source path is substituted for an argument when the [CallerFilePathAttribute](#) is applied to an optional parameter.
2. The source path is embedded in a PDB file.
3. The path of the PDB file is embedded into a PE (portable executable) file.

ApplicationConfiguration

The **ApplicationConfiguration** compiler option enables a C# application to specify the location of an assembly's application configuration (app.config) file to the common language runtime (CLR) at assembly binding time.

```
<ApplicationConfiguration>file</ApplicationConfiguration>
```

Where `file` is the application configuration file that contains assembly binding settings. One use of **ApplicationConfiguration** is advanced scenarios in which an assembly has to reference both the .NET Framework version and the .NET Framework for Silverlight version of a particular reference assembly at the same time. For example, a XAML designer written in Windows Presentation Foundation (WPF) might have to reference both the WPF Desktop, for the designer's user interface, and the subset of WPF that is included with Silverlight. The same designer assembly has to access both assemblies. By default, the separate references cause a compiler error, because assembly binding sees the two assemblies as equivalent. The **ApplicationConfiguration** compiler option enables you to specify the location of an app.config file that

disables the default behavior by using a `<supportPortability>` tag, as shown in the following example.

```
<supportPortability PKT="7cec85d7bea7798e" enable="false"/>
```

The compiler passes the location of the file to the CLR's assembly-binding logic.

NOTE

To use the app.config file that is already set in the project, add property tag `<UseAppConfigForCompiler>` to the .csproj file and set its value to `true`. To specify a different app.config file, add property tag `<AppConfigForCompiler>` and set its value to the location of the file.

The following example shows an app.config file that enables an application to have references to both the .NET Framework implementation and the .NET Framework for Silverlight implementation of any .NET Framework assembly that exists in both implementations. The **ApplicationConfiguration** compiler option specifies the location of this app.config file.

```
<configuration>
  <runtime>
    <assemblyBinding>
      <supportPortability PKT="7cec85d7bea7798e" enable="false"/>
      <supportPortability PKT="31bf3856ad364e35" enable="false"/>
    </assemblyBinding>
  </runtime>
</configuration>
```

AdditionalLibPaths

The **AdditionalLibPaths** option specifies the location of assemblies referenced with the **References** option.

```
<AdditionalLibPaths>dir1[,dir2]</AdditionalLibPaths>
```

Where `dir1` is a directory for the compiler to look in if a referenced assembly isn't found in the current working directory (the directory from which you're invoking the compiler) or in the common language runtime's system directory. `dir2` is one or more additional directories to search in for assembly references. Separate directory names with a comma, and without white space between them. The compiler searches for assembly references that aren't fully qualified in the following order:

1. Current working directory.
2. The common language runtime system directory.
3. Directories specified by **AdditionalLibPaths**.
4. Directories specified by the LIB environment variable.

Use **Reference** to specify an assembly reference. **AdditionalLibPaths** is additive. Specifying it more than once appends to any prior values. Since the path to the dependent assembly isn't specified in the assembly manifest, the application will find and use the assembly in the global assembly cache. The compiler referencing the assembly doesn't imply the common language runtime can find and load the assembly at run time. See [How the Runtime Locates Assemblies](#) for details on how the runtime searches for referenced assemblies.

GenerateFullPaths

The **GenerateFullPaths** option causes the compiler to specify the full path to the file when listing compilation errors and warnings.

```
<GenerateFullPaths>true</GenerateFullPaths>
```

By default, errors and warnings that result from compilation specify the name of the file in which an error was found. The **GenerateFullPaths** option causes the compiler to specify the full path to the file. This compiler option is unavailable in Visual Studio and cannot be changed programmatically.

PreferredUILang

By using the **PreferredUILang** compiler option, you can specify the language in which the C# compiler displays output, such as error messages.

```
<PreferredUILang>language</PreferredUILang>
```

Where `language` is the [language name](#) of the language to use for compiler output. You can use the **PreferredUILang** compiler option to specify the language that you want the C# compiler to use for error messages and other command-line output. If the language pack for the language isn't installed, the language setting of the operating system is used instead.

BaseAddress

The **BaseAddress** option lets you specify the preferred base address at which to load a DLL. For more information about when and why to use this option, see [Larry Osterman's WebLog](#).

```
<BaseAddress>address</BaseAddress>
```

Where `address` is the base address for the DLL. This address can be specified as a decimal, hexadecimal, or octal number. The default base address for a DLL is set by the .NET common language runtime. The lower-order word in this address will be rounded. For example, if you specify `0x11110001`, it will be rounded to `0x11110000`. To complete the signing process for a DLL, use SN.EXE with the -R option.

ChecksumAlgorithm

This option controls the checksum algorithm we use to encode source files in the PDB.

```
<ChecksumAlgorithm>algorithm</ChecksumAlgorithm>
```

The `algorithm` must be either `SHA1` (default) or `SHA256`.

CodePage

This option specifies which codepage to use during compilation if the required page isn't the current default codepage for the system.

```
<CodePage>id</CodePage>
```

Where `id` is the id of the code page to use for all source code files in the compilation. The compiler will first attempt to interpret all source files as UTF-8. If your source code files are in an encoding other than UTF-8 and use characters other than 7-bit ASCII characters, use the **CodePage** option to specify which code page should be used. **CodePage** applies to all source code files in your compilation. See [GetCPIInfo](#) for information on how to find which code pages are supported on your system.

Utf8Output

The **Utf8Output** option displays compiler output using UTF-8 encoding.

```
<Utf8Output>true</Utf8Output>
```

In some international configurations, compiler output cannot correctly be displayed in the console. Use **Utf8Output** and redirect compiler output to a file.

FileAlignment

The **FileAlignment** option lets you specify the size of sections in your output file. Valid values are 512, 1024, 2048, 4096, and 8192. These values are in bytes.

```
<FileAlignment>number</FileAlignment>
```

You set the **FileAlignment** option from the **Advanced** page of the **Build** properties for your project in Visual Studio. Each section will be aligned on a boundary that is a multiple of the **FileAlignment** value. There's no fixed default. If **FileAlignment** isn't specified, the common language runtime picks a default at compile time. By specifying the section size, you affect the size of the output file. Modifying section size may be useful for programs that will run on smaller devices. Use [DUMPBIN](#) to see information about sections in your output file.

ErrorEndLocation

Instructs the compiler to output line and column of the end location of each error.

```
<ErrorEndLocation>true</ErrorEndLocation>
```

By default, the compiler writes the starting location in source for all errors and warnings. When this option is set to true, the compiler writes both the starting and end location for each error and warning.

NoStandardLib

NoStandardLib prevents the import of mscorlib.dll, which defines the entire System namespace.

```
<NoStandardLib>true</NoStandardLib>
```

Use this option if you want to define or create your own System namespace and objects. If you don't specify **NoStandardLib**, mscorlib.dll is imported into your program (same as specifying

```
<NoStandardLib>>false</NoStandardLib> ).
```

SubsystemVersion

Specifies the minimum version of the subsystem on which the executable file runs. Most commonly, this option ensures that the executable file can use security features that aren't available with older versions of Windows.

NOTE

To specify the subsystem itself, use the [TargetType](#) compiler option.

```
<SubsystemVersion>major.minor</SubsystemVersion>
```

The `major.minor` specify the minimum required version of the subsystem, as expressed in a dot notation for major and minor versions. For example, you can specify that an application can't run on an operating system that's older than Windows 7. Set the value of this option to 6.01, as the table later in this article describes. You specify the values for `major` and `minor` as integers. Leading zeroes in the `minor` version don't change the version, but trailing zeroes do. For example, 6.1 and 6.01 refer to the same version, but 6.10 refers to a different version. We recommend expressing the minor version as two digits to avoid confusion.

The following table lists common subsystem versions of Windows.

WINDOWS VERSION	SUBSYSTEM VERSION
Windows Server 2003	5.02
Windows Vista	6.00
Windows 7	6.01
Windows Server 2008	6.01
Windows 8	6.02

The default value of the **SubsystemVersion** compiler option depends on the conditions in the following list:

- The default value is 6.02 if any compiler option in the following list is set:
 - [-target:appcontainerexe](#)
 - [-target:winmdobj](#)
 - [-platform:arm](#)
- The default value is 6.00 if you're using MSBuild, you're targeting .NET Framework 4.5, and you haven't set any of the compiler options that were specified earlier in this list.
- The default value is 4.00 if none of the previous conditions are true.

ModuleAssemblyName

Specifies the name of an assembly whose non-public types a *.netmodule* can access.

```
<ModuleAssemblyName>assembly_name</ModuleAssemblyName>
```

ModuleAssemblyName should be used when building a *.netmodule*, and where the following conditions are true:

- The *.netmodule* needs access to non-public types in an existing assembly.
- You know the name of the assembly into which the *.netmodule* will be built.
- The existing assembly has granted friend assembly access to the assembly into which the *.netmodule* will be built.

For more information on building a *.netmodule*, see [TargetType](#) option of **module**. For more information on friend assemblies, see [Friend Assemblies](#).

XML documentation comments

12/28/2021 • 11 minutes to read • [Edit Online](#)

C# source files can have structured comments that produce API documentation for the types defined in those files. The C# compiler produces an *XML* file that contains structured data representing the comments and the API signatures. Other tools can process that XML output to create human-readable documentation in the form of web pages or PDF files, for example.

This process provides many advantages for you to add API documentation in your code:

- The C# compiler combines the structure of the C# code with the text of the comments into a single XML document.
- The C# compiler verifies that the comments match the API signatures for relevant tags.
- Tools that process the XML documentation files can define XML elements and attributes specific to those tools.

Tools like Visual Studio provide IntelliSense for many common XML elements used in documentation comments.

This article covers these topics:

- Documentation comments and XML file generation
- Tags validated by the C# compiler and Visual Studio
- Format of the generated XML file

Create XML documentation output

You create documentation for your code by writing special comment fields indicated by triple slashes. The comment fields include XML elements that describe the code block that follows the comments. For example:

```
/// <summary>
/// This class performs an important function.
/// </summary>
public class MyClass {}
```

You set either the [GenerateDocumentationFile](#) or [DocumentationFile](#) option, and the compiler will find all comment fields with XML tags in the source code and create an XML documentation file from those comments. When this option is enabled, the compiler generates the [CS1591](#) warning for any publicly visible member declared in your project without XML documentation comments.

XML comment formats

The use of XML doc comments requires delimiters that indicate where a documentation comment begins and ends. You use the following delimiters with the XML documentation tags:

- `///` Single-line delimiter: The documentation examples and C# project templates use this form. If there's white space following the delimiter, it isn't included in the XML output.

NOTE

Visual Studio automatically inserts the `<summary>` and `</summary>` tags and positions your cursor within these tags after you type the `///` delimiter in the code editor. You can turn this feature on or off in the [Options dialog box](#).

- `/** */` Multiline delimiters: The `/** */` delimiters have the following formatting rules:
 - On the line that contains the `/**` delimiter, if the rest of the line is white space, the line isn't processed for comments. If the first character after the `/**` delimiter is white space, that white-space character is ignored and the rest of the line is processed. Otherwise, the entire text of the line after the `/**` delimiter is processed as part of the comment.
 - On the line that contains the `*/` delimiter, if there's only white space up to the `*/` delimiter, that line is ignored. Otherwise, the text on the line up to the `*/` delimiter is processed as part of the comment.
 - For the lines after the one that begins with the `/**` delimiter, the compiler looks for a common pattern at the beginning of each line. The pattern can consist of optional white space and an asterisk (`*`), followed by more optional white space. If the compiler finds a common pattern at the beginning of each line that doesn't begin with the `/**` delimiter or end with the `*/` delimiter, it ignores that pattern for each line.
 - The only part of the following comment that's processed is the line that begins with `<summary>`. The three tag formats produce the same comments.

```
/** <summary>text</summary> */  
  
/**  
<summary>text</summary>  
*/  
  
/**  
* <summary>text</summary>  
*/
```

- The compiler identifies a common pattern of " * " at the beginning of the second and third lines. The pattern isn't included in the output.

```
/**  
* <summary>  
* text </summary>*/
```

- The compiler finds no common pattern in the following comment because the second character on the third line isn't an asterisk. All text on the second and third lines is processed as part of the comment.

```
/**  
* <summary>  
  text </summary>  
*/
```

- The compiler finds no pattern in the following comment for two reasons. First, the number of spaces before the asterisk isn't consistent. Second, the fifth line begins with a tab, which doesn't match spaces. All text from lines two through five is processed as part of the comment.


```
/**
 * <summary>
 * text
 * text2
 * </summary>
 */
```

To refer to XML elements (for example, your function processes specific XML elements that you want to describe in an XML documentation comment), you can use the standard quoting mechanism (`<` and `>`). To refer to generic identifiers in code reference (`cref`) elements, you can use either the escape characters (for example, `cref="List<T>"`) or braces (`cref="List{T}"`). As a special case, the compiler parses the braces as angle brackets to make the documentation comment less cumbersome to author when referring to generic identifiers.

NOTE

The XML documentation comments are not metadata; they are not included in the compiled assembly and therefore they are not accessible through reflection.

Tools that accept XML documentation input

The following tools create output from XML comments:

- **DocFX:** *DocFX* is an API documentation generator for .NET, which currently supports C#, Visual Basic, and F#. It also allows you to customize the generated reference documentation. DocFX builds a static HTML website from your source code and Markdown files. Also, DocFX provides you the flexibility to customize the layout and style of your website through templates. You can also create custom templates.
- **Sandcastle:** The *Sandcastle tools* create help files for managed class libraries containing both conceptual and API reference pages. The Sandcastle tools are command-line based and have no GUI front-end, project management features, or automated build process. The Sandcastle Help File Builder provides standalone GUI and command-line based tools to build a help file in an automated fashion. A Visual Studio integration package is also available for it so that help projects can be created and managed entirely from within Visual Studio.
- **Doxygen:** *Doxygen* generates an on-line documentation browser (in HTML) or an off-line reference manual (in LaTeX) from a set of documented source files. There's also support for generating output in RTF (MS Word), PostScript, hyperlinked PDF, compressed HTML, DocBook, and Unix man pages. You can configure Doxygen to extract the code structure from undocumented source files.

ID strings

Each type or member is stored in an element in the output XML file. Each of those elements has a unique ID string that identifies the type or member. The ID string must account for operators, parameters, return values, generic type parameters, `ref`, `in`, and `out` parameters. To encode all those potential elements, the compiler follows clearly defined rules for generating the ID strings. Programs that process the XML file use the ID string to identify the corresponding .NET metadata or reflection item that the documentation applies to.

The compiler observes the following rules when it generates the ID strings:

- No white space is in the string.
- The first part of the string identifies the kind of member using a single character followed by a colon. The following member types are used:

CHARACTER	MEMBER TYPE	NOTES
N	namespace	You can't add documentation comments to a namespace, but you can make cref references to them, where supported.
T	type	A type is a class, interface, struct, enum, or delegate.
F	field	
P	property	Includes indexers or other indexed properties.
M	method	Includes special methods, such as constructors and operators.
E	event	
!	error string	The rest of the string provides information about the error. The C# compiler generates error information for links that cannot be resolved.

- The second part of the string is the fully qualified name of the item, starting at the root of the namespace. The name of the item, its enclosing type(s), and namespace are separated by periods. If the name of the item itself has periods, they're replaced by the hash-sign ('#'). It's assumed that no item has a hash-sign directly in its name. For example, the fully qualified name of the String constructor is "System.String.#ctor".
- For properties and methods, the parameter list enclosed in parentheses follows. If there are no parameters, no parentheses are present. The parameters are separated by commas. The encoding of each parameter follows directly how it's encoded in a .NET signature (See [Microsoft.VisualStudio.CoreDebugInterop.CorElementType](#) for definitions of the all caps elements in the following list):
 - Base types. Regular types (`ELEMENT_TYPE_CLASS` or `ELEMENT_TYPE_VALUETYPE`) are represented as the fully qualified name of the type.
 - Intrinsic types (for example, `ELEMENT_TYPE_I4`, `ELEMENT_TYPE_OBJECT`, `ELEMENT_TYPE_STRING`, `ELEMENT_TYPE_TYPEDBYREF`, and `ELEMENT_TYPE_VOID`) are represented as the fully qualified name of the corresponding full type. For example, `System.Int32` or `System.TypedReference`.
 - `ELEMENT_TYPE_PTR` is represented as a '*' following the modified type.
 - `ELEMENT_TYPE_BYREF` is represented as a '@' following the modified type.
 - `ELEMENT_TYPE_CMOD_OPT` is represented as a '!' and the fully qualified name of the modifier class, following the modified type.
 - `ELEMENT_TYPE_SZARRAY` is represented as "[]" following the element type of the array.
 - `ELEMENT_TYPE_ARRAY` is represented as [*lowerbound*:*size*,*lowerbound*:*size*] where the number of commas is the rank - 1, and the lower bounds and size of each dimension, if known, are represented in decimal. If a lower bound or size isn't specified, it's omitted. If the lower bound and size for a particular dimension are omitted, the ':' is omitted as well. For example, a two-dimensional array with 1 as the lower bounds and unspecified sizes is [1;1:].
- For conversion operators only (`op_Implicit` and `op_Explicit`), the return value of the method is

encoded as a `~` followed by the return type. For example:

`<member name="M:System.Decimal.op_Explicit(System.Decimal arg)~System.Int32">` is the tag for the cast operator `public static explicit operator int (decimal value);` declared in the `System.Decimal` class.

- For generic types, the name of the type is followed by a backtick and then a number that indicates the number of generic type parameters. For example: `<member name="T:SampleClass`2">` is the tag for a type that is defined as `public class SampleClass<T, U>`. For methods that take generic types as parameters, the generic type parameters are specified as numbers prefaced with backticks (for example ``0`,`1`). Each number represents a zero-based array notation for the type's generic parameters.
 - `ELEMENT_TYPE_PINNED` is represented as a `^` following the modified type. The C# compiler never generates this encoding.
 - `ELEMENT_TYPE_CMOD_REQ` is represented as a `|` and the fully qualified name of the modifier class, following the modified type. The C# compiler never generates this encoding.
 - `ELEMENT_TYPE_GENERICARRAY` is represented as `[?]` following the element type of the array. The C# compiler never generates this encoding.
 - `ELEMENT_TYPE_FNPTR` is represented as `=FUNC: type (signature)`, where `type` is the return type, and `signature` is the arguments of the method. If there are no arguments, the parentheses are omitted. The C# compiler never generates this encoding.
 - The following signature components aren't represented because they aren't used to differentiate overloaded methods:
 - calling convention
 - return type
 - `ELEMENT_TYPE_SENTINEL`

The following examples show how the ID strings for a class and its members are generated:

```
namespace MyNamespace
{
    /// <summary>
    /// Enter description here for class X.
    /// ID string generated is "T:MyNamespace.X".
    /// </summary>
    public unsafe class MyClass
    {
        /// <summary>
        /// Enter description here for the first constructor.
        /// ID string generated is "M:MyNamespace.MyClass.#ctor".
        /// </summary>
        public MyClass() { }

        /// <summary>
        /// Enter description here for the second constructor.
        /// ID string generated is "M:MyNamespace.MyClass.#ctor(System.Int32)".
        /// </summary>
        /// <param name="i">Describe parameter.</param>
        public MyClass(int i) { }

        /// <summary>
        /// Enter description here for field message.
        /// ID string generated is "F:MyNamespace.MyClass.message".
        /// </summary>
        public string message;

        /// <summary>
        /// Enter description for constant PI.
        /// ID string generated is "F:MyNamespace.MyClass.PI".
        /// </summary>
        public const double PI = 3.14;
    }
}
```

```

    /// <summary>
    /// Enter description for method func.
    /// ID string generated is "M:MyNamespace.MyClass.func".
    /// </summary>
    /// <returns>Describe return value.</returns>
    public int func() { return 1; }

    /// <summary>
    /// Enter description for method someMethod.
    /// ID string generated is
    "M:MyNamespace.MyClass.someMethod(System.String,System.Int32@,System.Void*)".
    /// </summary>
    /// <param name="str">Describe parameter.</param>
    /// <param name="num">Describe parameter.</param>
    /// <param name="ptr">Describe parameter.</param>
    /// <returns>Describe return value.</returns>
    public int someMethod(string str, ref int nm, void* ptr) { return 1; }

    /// <summary>
    /// Enter description for method anotherMethod.
    /// ID string generated is
    "M:MyNamespace.MyClass.anotherMethod(System.Int16[],System.Int32[0:,0:])".
    /// </summary>
    /// <param name="array1">Describe parameter.</param>
    /// <param name="array">Describe parameter.</param>
    /// <returns>Describe return value.</returns>
    public int anotherMethod(short[] array1, int[,] array) { return 0; }

    /// <summary>
    /// Enter description for operator.
    /// ID string generated is
    "M:MyNamespace.MyClass.op_Addition(MyNamespace.MyClass,MyNamespace.MyClass)".
    /// </summary>
    /// <param name="first">Describe parameter.</param>
    /// <param name="second">Describe parameter.</param>
    /// <returns>Describe return value.</returns>
    public static MyClass operator +(MyClass first, MyClass second) { return first; }

    /// <summary>
    /// Enter description for property.
    /// ID string generated is "P:MyNamespace.MyClass.prop".
    /// </summary>
    public int prop { get { return 1; } set { } }

    /// <summary>
    /// Enter description for event.
    /// ID string generated is "E:MyNamespace.MyClass.OnHappened".
    /// </summary>
    public event Del OnHappened;

    /// <summary>
    /// Enter description for index.
    /// ID string generated is "P:MyNamespace.MyClass.Item(System.String)".
    /// </summary>
    /// <param name="str">Describe parameter.</param>
    /// <returns></returns>
    public int this[string s] { get { return 1; } }

    /// <summary>
    /// Enter description for class Nested.
    /// ID string generated is "T:MyNamespace.MyClass.Nested".
    /// </summary>
    public class Nested { }

    /// <summary>
    /// Enter description for delegate.
    /// ID string generated is "T:MyNamespace.MyClass.Del".
    /// </summary>
    /// <param name="i">Describe parameter.</param>

```

```

/// <param name="myParameter">Describe parameter.</param>
public delegate void Del(int i);

/// <summary>
/// Enter description for operator.
/// ID string generated is "M:MyNamespace.MyClass.op_Explicit(MyNamespace.X)~System.Int32".
/// </summary>
/// <param name="myParameter">Describe parameter.</param>
/// <returns>Describe return value.</returns>
public static explicit operator int(MyClass myParameter) { return 1; }
}

```

C# language specification

For more information, see the [C# Language Specification](#) annex on documentation comments.

Recommended XML tags for C# documentation comments

12/28/2021 • 12 minutes to read • [Edit Online](#)

C# documentation comments use XML elements to define the structure of the output documentation. One consequence of this feature is that you can add any valid XML in your documentation comments. The C# compiler copies these elements into the output XML file. While you can use any valid XML in your comments (including any valid HTML element), documenting code is recommended for many reasons.

What follows are some recommendations, general use case scenarios, and things that you should know when using XML documentation tags in your C# code. While you can put any tags into your documentation comments, this article describes the recommended tags for the most common language constructs. In all cases, you should adhere to these recommendations:

- For the sake of consistency, all publicly visible types and their public members should be documented.
- Private members can also be documented using XML comments. However, it exposes the inner (potentially confidential) workings of your library.
- At a bare minimum, types and their members should have a `<summary>` tag because its content is needed for IntelliSense.
- Documentation text should be written using complete sentences ending with full stops.
- Partial classes are fully supported, and documentation information will be concatenated into a single entry for each type.

XML documentation starts with `///`. When you create a new project, the templates put some starter `///` lines in for you. The processing of these comments has some restrictions:

- The documentation must be well-formed XML. If the XML isn't well formed, the compiler generates a warning. The documentation file will contain a comment that says that an error was encountered.
- Some of the recommended tags have special meanings:
 - The `<param>` tag is used to describe parameters. If used, the compiler verifies that the parameter exists and that all parameters are described in the documentation. If the verification fails, the compiler issues a warning.
 - The `cref` attribute can be attached to any tag to reference a code element. The compiler verifies that this code element exists. If the verification fails, the compiler issues a warning. The compiler respects any `using` statements when it looks for a type described in the `cref` attribute.
 - The `<summary>` tag is used by IntelliSense inside Visual Studio to display additional information about a type or member.

NOTE

The XML file does not provide full information about the type and members (for example, it does not contain any type information). To get full information about a type or member, use the documentation file together with reflection on the actual type or member.

- Developers are free to create their own set of tags. The compiler will copy these to the output file.

Some of the recommended tags can be used on any language element. Others have more specialized usage. Finally, some of the tags are used to format text in your documentation. This article describes the recommended tags organized by their use.

The compiler verifies the syntax of the elements followed by a single * in the following list. Visual Studio provides IntelliSense for the tags verified by the compiler and all tags followed by ** in the following list. In addition to the tags listed here, the compiler and Visual Studio validate the ``, `<i>`, `<u>`, `
`, and `<a>` tags. The compiler also validates `<tt>`, which is deprecated HTML.

- **General Tags** used for multiple elements - These tags are the minimum set for any API.
 - `<summary>`: The value of this element is displayed in IntelliSense in Visual Studio.
 - `<remarks>` **
- **Tags used for members** - These tags are used when documenting methods and properties.
 - `<returns>`: The value of this element is displayed in IntelliSense in Visual Studio.
 - `<param>` *: The value of this element is displayed in IntelliSense in Visual Studio.
 - `<paramref>`
 - `<exception>` *
 - `<value>`: The value of this element is displayed in IntelliSense in Visual Studio.
- **Format documentation output** - These tags provide formatting directions for tools that generate documentation.
 - `<para>`
 - `<list>`
 - `<c>`
 - `<code>`
 - `<example>` **
- **Reuse documentation text** - These tags provide tools that make it easier to reuse XML comments.
 - `<inheritdoc>` **
 - `<include>` *
- **Generate links and references** - These tags generate links to other documentation.
 - `<see>` *
 - `<seealso>` *
 - `<cref>`
 - `<href>`
- **Tags for generic types and methods** - These tags are used only on generic types and methods
 - `<typeparam>` *: The value of this element is displayed in IntelliSense in Visual Studio.
 - `<typeparamref>`

NOTE

Documentation comments cannot be applied to a namespace.

If you want angle brackets to appear in the text of a documentation comment, use the HTML encoding of `<` and `>`, which is `<` and `>` respectively. This encoding is shown in the following example.

```
/// <summary>
/// This property always returns a value &lt; 1.
/// </summary>
```

General tags

<summary>

```
<summary>description</summary>
```

The `<summary>` tag should be used to describe a type or a type member. Use `<remarks>` to add supplemental information to a type description. Use the [cref attribute](#) to enable documentation tools such as [DocFX](#) and [Sandcastle](#) to create internal hyperlinks to documentation pages for code elements. The text for the `<summary>` tag is the only source of information about the type in IntelliSense, and is also displayed in the Object Browser window.

<remarks>

```
<remarks>  
description  
</remarks>
```

The `<remarks>` tag is used to add information about a type or a type member, supplementing the information specified with `<summary>`. This information is displayed in the Object Browser window. This tag may include more lengthy explanations. You may find that using `<CDATA>` sections for markdown make writing it more convenient. Tools such as [docfx](#) process the markdown text in `<CDATA>` sections.

Document members

<returns>

```
<returns>description</returns>
```

The `<returns>` tag should be used in the comment for a method declaration to describe the return value.

<param>

```
<param name="name">description</param>
```

- `name`: The name of a method parameter. Enclose the name in double quotation marks (" "). The names for parameters must match the API signature. If one or more parameter aren't covered, the compiler issues a warning. The compiler also issues a warning if the value of `name` doesn't match a formal parameter in the method declaration.

The `<param>` tag should be used in the comment for a method declaration to describe one of the parameters for the method. To document multiple parameters, use multiple `<param>` tags. The text for the `<param>` tag is displayed in IntelliSense, the Object Browser, and the Code Comment Web Report.

<paramref>

```
<paramref name="name"/>
```

- `name`: The name of the parameter to refer to. Enclose the name in double quotation marks (" ").

The `<paramref>` tag gives you a way to indicate that a word in the code comments, for example in a `<summary>` or `<remarks>` block refers to a parameter. The XML file can be processed to format this word in some distinct way, such as with a bold or italic font.

<exception>


```
<exception cref="member">description</exception>
```

- `cref = "member"`: A reference to an exception that is available from the current compilation environment. The compiler checks that the given exception exists and translates `member` to the canonical element name in the output XML. `member` must appear within double quotation marks (" ").

The `<exception>` tag lets you specify which exceptions can be thrown. This tag can be applied to definitions for methods, properties, events, and indexers.

<value>

```
<value>property-description</value>
```

The `<value>` tag lets you describe the value that a property represents. When you add a property via code wizard in the Visual Studio .NET development environment, it adds a `<summary>` tag for the new property. You manually add a `<value>` tag to describe the value that the property represents.

Format documentation output

<para>

```
<remarks>
  <para>
    This is an introductory paragraph.
  </para>
  <para>
    This paragraph contains more details.
  </para>
</remarks>
```

The `<para>` tag is for use inside a tag, such as `<summary>`, `<remarks>`, or `<returns>`, and lets you add structure to the text. The `<para>` tag creates a double spaced paragraph. Use the `
` tag if you want a single spaced paragraph.

<list>

```
<list type="bullet|number|table">
  <listheader>
    <term>term</term>
    <description>description</description>
  </listheader>
  <item>
    <term>Assembly</term>
    <description>The library or executable built from a compilation.</description>
  </item>
</list>
```

The `<listheader>` block is used to define the heading row of either a table or definition list. When defining a table, you only need to supply an entry for `term` in the heading. Each item in the list is specified with an `<item>` block. When creating a definition list, you'll need to specify both `term` and `description`. However, for a table, bulleted list, or numbered list, you only need to supply an entry for `description`. A list or table can have as many `<item>` blocks as needed.

<c>

```
<c>text</c>
```

The `<c>` tag gives you a way to indicate that text within a description should be marked as code. Use `<code>` to indicate multiple lines as code.

<code>

```
<code>
    var index = 5;
    index++;
</code>
```

The `<code>` tag is used to indicate multiple lines of code. Use `<c>` to indicate that single-line text within a description should be marked as code.

<example>

```
<example>
This shows how to increment an integer.
<code>
    var index = 5;
    index++;
</code>
</example>
```

The `<example>` tag lets you specify an example of how to use a method or other library member. An example commonly involves using the `<code>` tag.

Reuse documentation text

<inheritdoc>

```
<inheritdoc [cref=""] [path=""]/>
```

Inherit XML comments from base classes, interfaces, and similar methods. Using `inheritdoc` eliminates unwanted copying and pasting of duplicate XML comments and automatically keeps XML comments synchronized. Note that when you add the `<inheritdoc>` tag to a type, all members will inherit the comments as well.

- `cref`: Specify the member to inherit documentation from. Already defined tags on the current member are not overridden by the inherited ones.
- `path`: The XPath expression query that will result in a node set to show. You can use this attribute to filter the tags to include or exclude from the inherited documentation.

Add your XML comments in base classes or interfaces and let `inheritdoc` copy the comments to implementing classes. Add your XML comments to your synchronous methods and let `inheritdoc` copy the comments to your asynchronous versions of the same methods. If you want to copy the comments from a specific member, you use the `cref` attribute to specify the member.

<include>

```
<include file='filename' path='tagpath[@name="id"]' />
```

- `filename`: The name of the XML file containing the documentation. The file name can be qualified with a path

relative to the source code file. Enclose `filename` in single quotation marks (' ').

- `tagpath` : The path of the tags in `filename` that leads to the tag `name` . Enclose the path in single quotation marks (' ').
- `name` : The name specifier in the tag that precedes the comments; `name` will have an `id` .
- `id` : The ID for the tag that precedes the comments. Enclose the ID in double quotation marks (" ").

The `<include>` tag lets you refer to comments in another file that describe the types and members in your source code. Including an external file is an alternative to placing documentation comments directly in your source code file. By putting the documentation in a separate file, you can apply source control to the documentation separately from the source code. One person can have the source code file checked out and someone else can have the documentation file checked out. The `<include>` tag uses the XML XPath syntax. Refer to XPath documentation for ways to customize your `<include>` use.

Generate links and references

`<see>`

```
/// <see cref="member"/>
// or
/// <see cref="member">Link text</see>
// or
/// <see href="link">Link Text</see>
// or
/// <see langword="keyword"/>
```

- `cref="member"` : A reference to a member or field that is available to be called from the current compilation environment. The compiler checks that the given code element exists and passes `member` to the element name in the output XML. Place *member* within double quotation marks (" "). You can provide different link text for a "cref", by using a separate closing tag.
- `href="link"` : A clickable link to a given URL. For example, `<see href="https://github.com">GitHub</see>` produces a clickable link with text GitHub that links to `https://github.com` .
- `langword="keyword"` : A language keyword, such as `true` .

The `<see>` tag lets you specify a link from within text. Use `<seealso>` to indicate that text should be placed in a See Also section. Use the `cref attribute` to create internal hyperlinks to documentation pages for code elements. You include the type parameters to specify a reference to a generic type or method, such as `cref="IDictionary{T, U}"` . Also, `href` is a valid attribute that will function as a hyperlink.

`<seealso>`

```
/// <seealso cref="member"/>
// or
/// <seealso href="link">Link Text</seealso>
```

- `cref="member"` : A reference to a member or field that is available to be called from the current compilation environment. The compiler checks that the given code element exists and passes `member` to the element name in the output XML. `member` must appear within double quotation marks (" ").
- `href="link"` : A clickable link to a given URL. For example, `<seealso href="https://github.com">GitHub</seealso>` produces a clickable link with text GitHub that links to `https://github.com` .

The `<seealso>` tag lets you specify the text that you might want to appear in a **See Also** section. Use `<see>` to specify a link from within text. You cannot nest the `<seealso>` tag inside the `summary` tag.

cref attribute

The `cref` attribute in an XML documentation tag means "code reference." It specifies that the inner text of the tag is a code element, such as a type, method, or property. Documentation tools like [DocFX](#) and [Sandcastle](#) use the `cref` attributes to automatically generate hyperlinks to the page where the type or member is documented.

href attribute

The `href` attribute means a reference to a web page. You can use it to directly reference online documentation about your API or library.

Generic types and methods

<typeparam>

```
<typeparam name="TResult">The type returned from this method</typeparam>
```

- `TResult` : The name of the type parameter. Enclose the name in double quotation marks (" ").

The `<typeparam>` tag should be used in the comment for a generic type or method declaration to describe a type parameter. Add a tag for each type parameter of the generic type or method. The text for the `<typeparam>` tag will be displayed in IntelliSense.

<typeparamref>

```
<typeparamref name="TKey"/>
```

- `TKey` : The name of the type parameter. Enclose the name in double quotation marks (" ").

Use this tag to enable consumers of the documentation file to format the word in some distinct way, for example in italics.

User-defined tags

All the tags outlined above represent those tags that are recognized by the C# compiler. However, a user is free to define their own tags. Tools like Sandcastle bring support for extra tags like `<event>` and `<note>`, and even support [documenting namespaces](#). Custom or in-house documentation generation tools can also be used with the standard tags, and multiple output formats from HTML to PDF can be supported.

Example XML documentation comments

12/28/2021 • 18 minutes to read • [Edit Online](#)

This article contains three examples for adding XML documentation comments to most C# language elements. The first example shows how you document a class with different members. The second shows how you would reuse explanations for a hierarchy of classes or interfaces. The third shows tags to use for generic classes and members. The second and third examples use concepts that are covered in the first example.

Document a class, struct, or interface

The following example shows common language elements, and the tags you'll likely use to describe these elements. The documentation comments describe the use of the tags, rather than the class itself.

```
/// <summary>
/// Every class and member should have a one sentence
/// summary describing its purpose.
/// </summary>
/// <remarks>
/// You can expand on that one sentence summary to
/// provide more information for readers. In this case,
/// the <c>ExampleClass</c> provides different C#
/// elements to show how you would add documentation
/// comments for most elements in a typical class.
/// <para>
/// The remarks can add multiple paragraphs, so you can
/// write detailed information for developers that use
/// your work. You should add everything needed for
/// readers to be successful. This class contains
/// examples for the following:
/// </para>
/// <list type="table">
/// <item>
/// <term>Summary</term>
/// <description>
/// This should provide a one sentence summary of the class or member.
/// </description>
/// </item>
/// <item>
/// <term>Remarks</term>
/// <description>
/// This is typically a more detailed description of the class or member
/// </description>
/// </item>
/// <item>
/// <term>para</term>
/// <description>
/// The para tag separates a section into multiple paragraphs
/// </description>
/// </item>
/// <item>
/// <term>list</term>
/// <description>
/// Provides a list of terms or elements
/// </description>
/// </item>
/// <item>
/// <term>returns, param</term>
/// <description>
/// Used to describe parameters and return values
/// </description>
```

```

/// </item>
/// <item>
/// <term>value</term>
/// <description>Used to describe properties</description>
/// </item>
/// <item>
/// <term>exception</term>
/// <description>
/// Used to describe exceptions that may be thrown
/// </description>
/// </item>
/// <item>
/// <term>c, cref, see, seealso</term>
/// <description>
/// These provide code style and links to other
/// documentation elements
/// </description>
/// </item>
/// <item>
/// <term>example, code</term>
/// <description>
/// These are used for code examples
/// </description>
/// </item>
/// </list>
/// <para>
/// The list above uses the "table" style. You could
/// also use the "bullet" or "number" style. Neither
/// would typically use the "term" element.
/// <br/>
/// Note: paragraphs are double spaced. Use the *br*
/// tag for single spaced lines.
/// </para>
/// </remarks>
public class ExampleClass
{
    /// <value>
    /// The <c>Label</c> property represents a label
    /// for this instance.
    /// </value>
    /// <remarks>
    /// The <see cref="Label"/> is a <see langword="string"/>
    /// that you use for a label.
    /// <para>
    /// Note that there isn't a way to provide a "cref" to
    /// each accessor, only to the property itself.
    /// </para>
    /// </remarks>
    public string Label
    {
        get;
        set;
    }

    /// <summary>
    /// Adds two integers and returns the result.
    /// </summary>
    /// <returns>
    /// The sum of two integers.
    /// </returns>
    /// <param name="left">
    /// The left operand of the addition.
    /// </param>
    /// <param name="right">
    /// The right operand of the addition.
    /// </param>
    /// <example>
    /// <code>
    /// int c = Math.Add(4, 5);

```

```

    /// if (c > 10)
    /// {
    ///     Console.WriteLine(c);
    /// }
    /// </code>
    /// </example>
    /// <exception cref="System.OverflowException">
    /// Thrown when one parameter is
    /// <see cref="Int32.MaxValue">MaxValue</see> and the other is
    /// greater than 0.
    /// Note that here you can also use
    /// <see href="https://docs.microsoft.com/dotnet/api/system.int32.maxvalue"/>
    /// to point a web page instead.
    /// </exception>
    /// <see cref="ExampleClass"/> for a list of all
    /// the tags in these examples.
    /// <seealso cref="ExampleClass.Label"/>
    public static int Add(int left, int right)
    {
        if ((left == int.MaxValue && right > 0) || (right == int.MaxValue && left > 0))
            throw new System.OverflowException();

        return left + right;
    }
}

    /// <summary>
    /// This is an example of a positional record.
    /// </summary>
    /// <remarks>
    /// There isn't a way to add XML comments for properties
    /// created for positional records, yet. The language
    /// design team is still considering what tags should
    /// be supported, and where. Currently, you can use
    /// the "param" tag to describe the parameters to the
    /// primary constructor.
    /// </remarks>
    /// <param name="FirstName">
    /// This tag will apply to the primary constructor parameter.
    /// </param>
    /// <param name="LastName">
    /// This tag will apply to the primary constructor parameter.
    /// </param>
    public record Person(string FirstName, string LastName);
}

```

Adding documentation can clutter your source code with large sets of comments intended for users of your library. You use the `<Include>` tag to separate your XML comments from your source. Your source code references an XML file with the `<Include>` tag:

```

    /// <include file='xml_include_tag.xml' path='MyDocs/MyMembers[@name="test"]/*' />
    class Test
    {
        static void Main()
        {
        }
    }

    /// <include file='xml_include_tag.xml' path='MyDocs/MyMembers[@name="test2"]/*' />
    class Test2
    {
        public void Test()
        {
        }
    }
}

```

The second file, *xml_include_tag.xml*, contains the documentation comments.

```
<MyDocs>
  <MyMembers name="test">
    <summary>
      The summary for this type.
    </summary>
  </MyMembers>
  <MyMembers name="test2">
    <summary>
      The summary for this other type.
    </summary>
  </MyMembers>
</MyDocs>
```

Document a hierarchy of classes and interfaces

The `<inheritdoc>` element means a type or member *inherits* documentation comments from a base class or interface. You can also use the `<inheritdoc>` element with the `cref` attribute to inherit comments from a member of the same type. The following example shows ways to use this tag. Note that when you add the `inheritdoc` attribute to a type, member comments are inherited. You can prevent the use of inherited comments by writing comments on the members in the derived type. Those will be chosen over the inherited comments.

```
/// <summary>
/// A summary about this class.
/// </summary>
/// <remarks>
/// These remarks would explain more about this class.
/// In this example, these comments also explain the
/// general information about the derived class.
/// </remarks>
public class MainClass
{
}

///<inheritdoc/>
public class DerivedClass : MainClass
{
}

/// <summary>
/// This interface would describe all the methods in
/// its contract.
/// </summary>
/// <remarks>
/// While elided for brevity, each method or property
/// in this interface would contain docs that you want
/// to duplicate in each implementing class.
/// </remarks>
public interface ITestInterface
{
  /// <summary>
  /// This method is part of the test interface.
  /// </summary>
  /// <remarks>
  /// This content would be inherited by classes
  /// that implement this interface when the
  /// implementing class uses "inheritdoc"
  /// </remarks>
  /// <returns>The value of <paramref name="arg" /> </returns>
  /// <param name="arg">The argument to the method</param>
  int Method(int arg);
}
```



```

}

///
public class ImplementingClass : ITestInterface
{
    // doc comments are inherited here.
    public int Method(int arg) => arg;
}

/// <summary>
/// This class shows hows you can "inherit" the doc
/// comments from one method in another method.
/// </summary>
/// <remarks>
/// You can inherit all comments, or only a specific tag,
/// represented by an xpath expression.
/// </remarks>
public class InheritOnlyReturns
{
    /// <summary>
    /// In this example, this summary is only visible for this method.
    /// </summary>
    /// <returns>A boolean</returns>
    public static bool MyParentMethod(bool x) { return x; }

    /// <inheritdoc cref="MyParentMethod" path="/returns"/>
    public static bool MyChildMethod() { return false; }
}

/// <Summary>
/// This class shows an example ofsharing comments across methods.
/// </Summary>
public class InheritAllButRemarks
{
    /// <summary>
    /// In this example, this summary is visible on all the methods.
    /// </summary>
    /// <remarks>
    /// The remarks can be inherited by other methods
    /// using the xpath expression.
    /// </remarks>
    /// <returns>A boolean</returns>
    public static bool MyParentMethod(bool x) { return x; }

    /// <inheritdoc cref="MyParentMethod" path="//*[not(self::remarks)]"/>
    public static bool MyChildMethod() { return false; }
}

```

Generic types

Use the `<typeparam>` tag to describe type parameters on generic types and methods. The value for the `cref` attribute requires new syntax to reference a generic method or class:

```

/// <summary>
/// This is a generic class.
/// </summary>
/// <remarks>
/// This example shows how to specify the <see cref="GenericClass{T}"/>
/// type as a cref attribute.
/// In generic classes and methods, you'll often want to reference the
/// generic type, or the type parameter.
/// </remarks>
class GenericClass<T>
{
    // Fields and members.
}

/// <Summary>
/// This shows examples of typeparamref and typeparam tags
/// </Summary>
public class ParamsAndParamRefs
{
    /// <summary>
    /// The GetGenericValue method.
    /// </summary>
    /// <remarks>
    /// This sample shows how to specify the <see cref="GetGenericValue"/>
    /// method as a cref attribute.
    /// The parameter and return value are both of an arbitrary type,
    /// <typeparamref name="T"/>
    /// </remarks>
    public static T GetGenericValue<T>(T para)
    {
        return para;
    }
}

```

Math class example

The following code shows a realistic example of adding doc comments to a math library.

```

namespace TaggedLibrary
{
    /*
        The main Math class
        Contains all methods for performing basic math functions
    */
    /// <summary>
    /// The main <c>Math</c> class.
    /// Contains all methods for performing basic math functions.
    /// <list type="bullet">
    /// <item>
    /// <term>Add</term>
    /// <description>Addition Operation</description>
    /// </item>
    /// <item>
    /// <term>Subtract</term>
    /// <description>Subtraction Operation</description>
    /// </item>
    /// <item>
    /// <term>Multiply</term>
    /// <description>Multiplication Operation</description>
    /// </item>
    /// <item>
    /// <term>Divide</term>
    /// <description>Division Operation</description>
    /// </item>
    /// </list>

```

```

/// </summary>
/// <remarks>
/// <para>
/// This class can add, subtract, multiply and divide.
/// </para>
/// <para>
/// These operations can be performed on both
/// integers and doubles.
/// </para>
/// </remarks>
public class Math
{
    // Adds two integers and returns the result
    /// <summary>
    /// Adds two integers <paramref name="a"/> and <paramref name="b"/>
    /// and returns the result.
    /// </summary>
    /// <returns>
    /// The sum of two integers.
    /// </returns>
    /// <example>
    /// <code>
    /// int c = Math.Add(4, 5);
    /// if (c > 10)
    /// {
    ///     Console.WriteLine(c);
    /// }
    /// </code>
    /// </example>
    /// <exception cref="System.OverflowException">
    /// Thrown when one parameter is <see cref="Int32.MaxValue"/> and the other
    /// is greater than 0.
    /// </exception>
    /// See <see cref="Math.Add(double, double)"/> to add doubles.
    /// <seealso cref="Math.Subtract(int, int)"/>
    /// <seealso cref="Math.Multiply(int, int)"/>
    /// <seealso cref="Math.Divide(int, int)"/>
    /// <param name="a">An integer.</param>
    /// <param name="b">An integer.</param>
    public static int Add(int a, int b)
    {
        // If any parameter is equal to the max value of an integer
        // and the other is greater than zero
        if ((a == int.MaxValue && b > 0) ||
            (b == int.MaxValue && a > 0))
        {
            throw new System.OverflowException();
        }
        return a + b;
    }

    // Adds two doubles and returns the result
    /// <summary>
    /// Adds two doubles <paramref name="a"/> and <paramref name="b"/>
    /// and returns the result.
    /// </summary>
    /// <returns>
    /// The sum of two doubles.
    /// </returns>
    /// <example>
    /// <code>
    /// double c = Math.Add(4.5, 5.4);
    /// if (c > 10)
    /// {
    ///     Console.WriteLine(c);
    /// }
    /// </code>
    /// </example>
    /// <exception cref="System.OverflowException">

```

```

/// Thrown when one parameter is max and the other
/// is greater than 0.</exception>
/// See <see cref="Math.Add(int, int)"/> to add integers.
/// <seealso cref="Math.Subtract(double, double)"/>
/// <seealso cref="Math.Multiply(double, double)"/>
/// <seealso cref="Math.Divide(double, double)"/>
/// <param name="a">A double precision number.</param>
/// <param name="b">A double precision number.</param>
public static double Add(double a, double b)
{
    // If any parameter is equal to the max value of an integer
    // and the other is greater than zero
    if ((a == double.MaxValue && b > 0)
        || (b == double.MaxValue && a > 0))
    {
        throw new System.OverflowException();
    }

    return a + b;
}

// Subtracts an integer from another and returns the result
/// <summary>
/// Subtracts <paramref name="b"/> from <paramref name="a"/>
/// and returns the result.
/// </summary>
/// <returns>
/// The difference between two integers.
/// </returns>
/// <example>
/// <code>
/// int c = Math.Subtract(4, 5);
/// if (c > 1)
/// {
///     Console.WriteLine(c);
/// }
/// </code>
/// </example>
/// See <see cref="Math.Subtract(double, double)"/> to subtract doubles.
/// <seealso cref="Math.Add(int, int)"/>
/// <seealso cref="Math.Multiply(int, int)"/>
/// <seealso cref="Math.Divide(int, int)"/>
/// <param name="a">An integer.</param>
/// <param name="b">An integer.</param>
public static int Subtract(int a, int b)
{
    return a - b;
}

// Subtracts a double from another and returns the result
/// <summary>
/// Subtracts a double <paramref name="b"/> from another
/// double <paramref name="a"/> and returns the result.
/// </summary>
/// <returns>
/// The difference between two doubles.
/// </returns>
/// <example>
/// <code>
/// double c = Math.Subtract(4.5, 5.4);
/// if (c > 1)
/// {
///     Console.WriteLine(c);
/// }
/// </code>
/// </example>
/// See <see cref="Math.Subtract(int, int)"/> to subtract integers.
/// <seealso cref="Math.Add(double, double)"/>
/// <seealso cref="Math.Multiply(double, double)"/>

```

```

/// <seealso cref="Math.Divide(double, double)"/>
/// <summary>
/// <param name="a">A double precision number.</param>
/// <param name="b">A double precision number.</param>
public static double Subtract(double a, double b)
{
    return a - b;
}

// Multiplies two integers and returns the result
/// <summary>
/// Multiplies two integers <paramref name="a"/>
/// and <paramref name="b"/> and returns the result.
/// </summary>
/// <returns>
/// The product of two integers.
/// </returns>
/// <example>
/// <code>
/// int c = Math.Multiply(4, 5);
/// if (c > 100)
/// {
///     Console.WriteLine(c);
/// }
/// </code>
/// </example>
/// See <see cref="Math.Multiply(double, double)"/> to multiply doubles.
/// <seealso cref="Math.Add(int, int)"/>
/// <seealso cref="Math.Subtract(int, int)"/>
/// <seealso cref="Math.Divide(int, int)"/>
/// <param name="a">An integer.</param>
/// <param name="b">An integer.</param>
public static int Multiply(int a, int b)
{
    return a * b;
}

// Multiplies two doubles and returns the result
/// <summary>
/// Multiplies two doubles <paramref name="a"/> and
/// <paramref name="b"/> and returns the result.
/// </summary>
/// <returns>
/// The product of two doubles.
/// </returns>
/// <example>
/// <code>
/// double c = Math.Multiply(4.5, 5.4);
/// if (c > 100.0)
/// {
///     Console.WriteLine(c);
/// }
/// </code>
/// </example>
/// See <see cref="Math.Multiply(int, int)"/> to multiply integers.
/// <seealso cref="Math.Add(double, double)"/>
/// <seealso cref="Math.Subtract(double, double)"/>
/// <seealso cref="Math.Divide(double, double)"/>
/// <param name="a">A double precision number.</param>
/// <param name="b">A double precision number.</param>
public static double Multiply(double a, double b)
{
    return a * b;
}

// Divides an integer by another and returns the result
/// <summary>
/// Divides an integer <paramref name="a"/> by another
/// integer <paramref name="b"/> and returns the result.
/// </summary>

```

```

    /// </summary>
    /// <returns>
    /// The quotient of two integers.
    /// </returns>
    /// <example>
    /// <code>
    /// int c = Math.Divide(4, 5);
    /// if (c > 1)
    /// {
    ///     Console.WriteLine(c);
    /// }
    /// </code>
    /// </example>
    /// <exception cref="System.DivideByZeroException">
    /// Thrown when <paramref name="b"/> is equal to 0.
    /// </exception>
    /// See <see cref="Math.Divide(double, double)"/> to divide doubles.
    /// <seealso cref="Math.Add(int, int)"/>
    /// <seealso cref="Math.Subtract(int, int)"/>
    /// <seealso cref="Math.Multiply(int, int)"/>
    /// <param name="a">An integer dividend.</param>
    /// <param name="b">An integer divisor.</param>
    public static int Divide(int a, int b)
    {
        return a / b;
    }

    // Divides a double by another and returns the result
    /// <summary>
    /// Divides a double <paramref name="a"/> by another double
    /// <paramref name="b"/> and returns the result.
    /// </summary>
    /// <returns>
    /// The quotient of two doubles.
    /// </returns>
    /// <example>
    /// <code>
    /// double c = Math.Divide(4.5, 5.4);
    /// if (c > 1.0)
    /// {
    ///     Console.WriteLine(c);
    /// }
    /// </code>
    /// </example>
    /// <exception cref="System.DivideByZeroException">
    /// Thrown when <paramref name="b"/> is equal to 0.
    /// </exception>
    /// See <see cref="Math.Divide(int, int)"/> to divide integers.
    /// <seealso cref="Math.Add(double, double)"/>
    /// <seealso cref="Math.Subtract(double, double)"/>
    /// <seealso cref="Math.Multiply(double, double)"/>
    /// <param name="a">A double precision dividend.</param>
    /// <param name="b">A double precision divisor.</param>
    public static double Divide(double a, double b)
    {
        return a / b;
    }
}

```

You may find that the code is obscured by all the comments. The final example shows how you would adapt this library to use the `include` tag. You move all the documentation to an XML file:

```

<docs>
  <members name="math">
    <Math>
      <summary>

```

```

The main <c>Math</c> class.
Contains all methods for performing basic math functions.
</summary>
<remarks>
<para>This class can add, subtract, multiply and divide.</para>
<para>These operations can be performed on both integers and doubles.</para>
</remarks>
</Math>
<AddInt>
<summary>
Adds two integers <paramref name="a"/> and <paramref name="b"/>
and returns the result.
</summary>
<returns>
The sum of two integers.
</returns>
<example>
<code>
int c = Math.Add(4, 5);
if (c > 10)
{
    Console.WriteLine(c);
}
</code>
</example>
<exception cref="System.OverflowException">Thrown when one
parameter is max
and the other is greater than 0.</exception>
See <see cref="Math.Add(double, double)"/> to add doubles.
<seealso cref="Math.Subtract(int, int)"/>
<seealso cref="Math.Multiply(int, int)"/>
<seealso cref="Math.Divide(int, int)"/>
<param name="a">An integer.</param>
<param name="b">An integer.</param>
</AddInt>
<AddDouble>
<summary>
Adds two doubles <paramref name="a"/> and <paramref name="b"/>
and returns the result.
</summary>
<returns>
The sum of two doubles.
</returns>
<example>
<code>
double c = Math.Add(4.5, 5.4);
if (c > 10)
{
    Console.WriteLine(c);
}
</code>
</example>
<exception cref="System.OverflowException">Thrown when one parameter is max
and the other is greater than 0.</exception>
See <see cref="Math.Add(int, int)"/> to add integers.
<seealso cref="Math.Subtract(double, double)"/>
<seealso cref="Math.Multiply(double, double)"/>
<seealso cref="Math.Divide(double, double)"/>
<param name="a">A double precision number.</param>
<param name="b">A double precision number.</param>
</AddDouble>
<SubtractInt>
<summary>
Subtracts <paramref name="b"/> from <paramref name="a"/> and
returns the result.
</summary>
<returns>
The difference between two integers.
</returns>

```

```

<example>
<code>
int c = Math.Subtract(4, 5);
if (c > 1)
{
    Console.WriteLine(c);
}
</code>
</example>
See <see cref="Math.Subtract(double, double)"/> to subtract doubles.
<seealso cref="Math.Add(int, int)"/>
<seealso cref="Math.Multiply(int, int)"/>
<seealso cref="Math.Divide(int, int)"/>
<param name="a">An integer.</param>
<param name="b">An integer.</param>
</SubtractInt>
<SubtractDouble>
<summary>
Subtracts a double <paramref name="b"/> from another
double <paramref name="a"/> and returns the result.
</summary>
<returns>
The difference between two doubles.
</returns>
<example>
<code>
double c = Math.Subtract(4.5, 5.4);
if (c > 1)
{
    Console.WriteLine(c);
}
</code>
</example>
See <see cref="Math.Subtract(int, int)"/> to subtract integers.
<seealso cref="Math.Add(double, double)"/>
<seealso cref="Math.Multiply(double, double)"/>
<seealso cref="Math.Divide(double, double)"/>
<param name="a">A double precision number.</param>
<param name="b">A double precision number.</param>
</SubtractDouble>
<MultiplyInt>
<summary>
Multiplies two integers <paramref name="a"/> and
<paramref name="b"/> and returns the result.
</summary>
<returns>
The product of two integers.
</returns>
<example>
<code>
int c = Math.Multiply(4, 5);
if (c > 100)
{
    Console.WriteLine(c);
}
</code>
</example>
See <see cref="Math.Multiply(double, double)"/> to multiply doubles.
<seealso cref="Math.Add(int, int)"/>
<seealso cref="Math.Subtract(int, int)"/>
<seealso cref="Math.Divide(int, int)"/>
<param name="a">An integer.</param>
<param name="b">An integer.</param>
</MultiplyInt>
<MultiplyDouble>
<summary>
Multiplies two doubles <paramref name="a"/> and
<paramref name="b"/> and returns the result.
</summary>

```



```

<returns>
The product of two doubles.
</returns>
<example>
<code>
double c = Math.Multiply(4.5, 5.4);
if (c > 100.0)
{
    Console.WriteLine(c);
}
</code>
</example>
See <see cref="Math.Multiply(int, int)"/> to multiply integers.
<seealso cref="Math.Add(double, double)"/>
<seealso cref="Math.Subtract(double, double)"/>
<seealso cref="Math.Divide(double, double)"/>
<param name="a">A double precision number.</param>
<param name="b">A double precision number.</param>
</MultiplyDouble>
<DivideInt>
<summary>
Divides an integer <paramref name="a"/> by another integer
<paramref name="b"/> and returns the result.
</summary>
<returns>
The quotient of two integers.
</returns>
<example>
<code>
int c = Math.Divide(4, 5);
if (c > 1)
{
    Console.WriteLine(c);
}
</code>
</example>
<exception cref="System.DivideByZeroException">
Thrown when <paramref name="b"/> is equal to 0.
</exception>
See <see cref="Math.Divide(double, double)"/> to divide doubles.
<seealso cref="Math.Add(int, int)"/>
<seealso cref="Math.Subtract(int, int)"/>
<seealso cref="Math.Multiply(int, int)"/>
<param name="a">An integer dividend.</param>
<param name="b">An integer divisor.</param>
</DivideInt>
<DivideDouble>
<summary>
Divides a double <paramref name="a"/> by another
double <paramref name="b"/> and returns the result.
</summary>
<returns>
The quotient of two doubles.
</returns>
<example>
<code>
double c = Math.Divide(4.5, 5.4);
if (c > 1.0)
{
    Console.WriteLine(c);
}
</code>
</example>
<exception cref="System.DivideByZeroException">Thrown when <paramref name="b"/> is equal to 0.
</exception>
See <see cref="Math.Divide(int, int)"/> to divide integers.
<seealso cref="Math.Add(double, double)"/>
<seealso cref="Math.Subtract(double, double)"/>
<seealso cref="Math.Multiply(double, double)"/>

```

```

</param>
</param>
</DivideDouble>
</members>
</docs>

```

In the above XML, each member's documentation comments appear directly inside a tag named after what they do. You can choose your own strategy. The code uses the `<include>` tag to reference the appropriate element in the XML file:

```

namespace IncludeTag
{
    /*
        The main Math class
        Contains all methods for performing basic math functions
    */
    /// <include file='include.xml' path='docs/members[@name="math"]/Math/'/>
    public class Math
    {
        // Adds two integers and returns the result
        /// <include file='include.xml' path='docs/members[@name="math"]/AddInt/'/>
        public static int Add(int a, int b)
        {
            // If any parameter is equal to the max value of an integer
            // and the other is greater than zero
            if ((a == int.MaxValue && b > 0) || (b == int.MaxValue && a > 0))
                throw new System.OverflowException();

            return a + b;
        }

        // Adds two doubles and returns the result
        /// <include file='include.xml' path='docs/members[@name="math"]/AddDouble/'/>
        public static double Add(double a, double b)
        {
            // If any parameter is equal to the max value of an integer
            // and the other is greater than zero
            if ((a == double.MaxValue && b > 0) || (b == double.MaxValue && a > 0))
                throw new System.OverflowException();

            return a + b;
        }

        // Subtracts an integer from another and returns the result
        /// <include file='include.xml' path='docs/members[@name="math"]/SubtractInt/'/>
        public static int Subtract(int a, int b)
        {
            return a - b;
        }

        // Subtracts a double from another and returns the result
        /// <include file='include.xml' path='docs/members[@name="math"]/SubtractDouble/'/>
        public static double Subtract(double a, double b)
        {
            return a - b;
        }

        // Multiplies two integers and returns the result
        /// <include file='include.xml' path='docs/members[@name="math"]/MultiplyInt/'/>
        public static int Multiply(int a, int b)
        {
            return a * b;
        }

        // Multiplies two doubles and returns the result
    }
}

```

```

    /// <include file='include.xml' path='docs/members[@name="math"]/MultiplyDouble/*' />
    public static double Multiply(double a, double b)
    {
        return a * b;
    }

    // Divides an integer by another and returns the result
    /// <include file='include.xml' path='docs/members[@name="math"]/DivideInt/*' />
    public static int Divide(int a, int b)
    {
        return a / b;
    }

    // Divides a double by another and returns the result
    /// <include file='include.xml' path='docs/members[@name="math"]/DivideDouble/*' />
    public static double Divide(double a, double b)
    {
        return a / b;
    }
}

```

- The `file` attribute represents the name of the XML file containing the documentation.
- The `path` attribute represents an [XPath](#) query to the `tag name` present in the specified `file`.
- The `name` attribute represents the name specifier in the tag that precedes the comments.
- The `id` attribute, which can be used in place of `name`, represents the ID for the tag that precedes the comments.

C# Compiler Errors

12/28/2021 • 2 minutes to read • [Edit Online](#)

Some C# compiler errors have corresponding topics that explain why the error is generated, and, in some cases, how to fix the error. Use one of the following steps to see whether help is available for a particular error message.

- If you're using Visual Studio, choose the error number (for example, CS0029) in the [Output Window](#), and then choose the F1 key.
- Type the error number in the *Filter by title* box in the table of contents.

If none of these steps leads to information about your error, go to the end of this page, and send feedback that includes the number or text of the error.

For information about how to configure error and warning options in C#, see [C# compiler options](#) or the Visual Studio [Build Page, Project Designer \(C#\)](#).

NOTE

Your computer might show different names or locations for some of the Visual Studio user interface elements in the following instructions. The Visual Studio edition that you have and the settings that you use determine these elements. For more information, see [Personalizing the IDE](#).

See also

- [C# Compiler Options](#)
- [Build Page, Project Designer \(C#\)](#)
- [WarningLevel \(C# Compiler Options\)](#)
- [DisabledWarnings \(C# Compiler Options\)](#)

Introduction

12/28/2021 • 55 minutes to read • [Edit Online](#)

C# (pronounced "See Sharp") is a simple, modern, object-oriented, and type-safe programming language. C# has its roots in the C family of languages and will be immediately familiar to C, C++, and Java programmers. C# is standardized by ECMA International as the *ECMA-334* standard and by ISO/IEC as the *ISO/IEC 23270* standard. Microsoft's C# compiler for the .NET Framework is a conforming implementation of both of these standards.

C# is an object-oriented language, but C# further includes support for *component-oriented* programming. Contemporary software design increasingly relies on software components in the form of self-contained and self-describing packages of functionality. Key to such components is that they present a programming model with properties, methods, and events; they have attributes that provide declarative information about the component; and they incorporate their own documentation. C# provides language constructs to directly support these concepts, making C# a very natural language in which to create and use software components.

Several C# features aid in the construction of robust and durable applications: *Garbage collection* automatically reclaims memory occupied by unused objects; *exception handling* provides a structured and extensible approach to error detection and recovery; and the *type-safe* design of the language makes it impossible to read from uninitialized variables, to index arrays beyond their bounds, or to perform unchecked type casts.

C# has a *unified type system*. All C# types, including primitive types such as `int` and `double`, inherit from a single root `object` type. Thus, all types share a set of common operations, and values of any type can be stored, transported, and operated upon in a consistent manner. Furthermore, C# supports both user-defined reference types and value types, allowing dynamic allocation of objects as well as in-line storage of lightweight structures.

To ensure that C# programs and libraries can evolve over time in a compatible manner, much emphasis has been placed on *versioning* in C#'s design. Many programming languages pay little attention to this issue, and, as a result, programs written in those languages break more often than necessary when newer versions of dependent libraries are introduced. Aspects of C#'s design that were directly influenced by versioning considerations include the separate `virtual` and `override` modifiers, the rules for method overload resolution, and support for explicit interface member declarations.

The rest of this chapter describes the essential features of the C# language. Although later chapters describe rules and exceptions in a detail-oriented and sometimes mathematical manner, this chapter strives for clarity and brevity at the expense of completeness. The intent is to provide the reader with an introduction to the language that will facilitate the writing of early programs and the reading of later chapters.

Hello world

The "Hello, World" program is traditionally used to introduce a programming language. Here it is in C#:

```
using System;

class Hello
{
    static void Main() {
        Console.WriteLine("Hello, World");
    }
}
```

C# source files typically have the file extension `.cs`. Assuming that the "Hello, World" program is stored in the file `hello.cs`, the program can be compiled with the Microsoft C# compiler using the command line

```
csc hello.cs
```

which produces an executable assembly named `hello.exe`. The output produced by this application when it is run is

```
Hello, World
```

The "Hello, World" program starts with a `using` directive that references the `System` namespace. Namespaces provide a hierarchical means of organizing C# programs and libraries. Namespaces contain types and other namespaces—for example, the `System` namespace contains a number of types, such as the `Console` class referenced in the program, and a number of other namespaces, such as `IO` and `Collections`. A `using` directive that references a given namespace enables unqualified use of the types that are members of that namespace. Because of the `using` directive, the program can use `Console.WriteLine` as shorthand for `System.Console.WriteLine`.

The `Hello` class declared by the "Hello, World" program has a single member, the method named `Main`. The `Main` method is declared with the `static` modifier. While instance methods can reference a particular enclosing object instance using the keyword `this`, static methods operate without reference to a particular object. By convention, a static method named `Main` serves as the entry point of a program.

The output of the program is produced by the `WriteLine` method of the `Console` class in the `System` namespace. This class is provided by the .NET Framework class libraries, which, by default, are automatically referenced by the Microsoft C# compiler. Note that C# itself does not have a separate runtime library. Instead, the .NET Framework is the runtime library of C#.

Program structure

The key organizational concepts in C# are *programs*, *namespaces*, *types*, *members*, and *assemblies*. C# programs consist of one or more source files. Programs declare types, which contain members and can be organized into namespaces. Classes and interfaces are examples of types. Fields, methods, properties, and events are examples of members. When C# programs are compiled, they are physically packaged into assemblies. Assemblies typically have the file extension `.exe` or `.dll`, depending on whether they implement *applications* or *libraries*.

The example

```

using System;

namespace Acme.Collections
{
    public class Stack
    {
        Entry top;

        public void Push(object data) {
            top = new Entry(top, data);
        }

        public object Pop() {
            if (top == null) throw new InvalidOperationException();
            object result = top.data;
            top = top.next;
            return result;
        }

        class Entry
        {
            public Entry next;
            public object data;

            public Entry(Entry next, object data) {
                this.next = next;
                this.data = data;
            }
        }
    }
}

```

declares a class named `Stack` in a namespace called `Acme.Collections`. The fully qualified name of this class is `Acme.Collections.Stack`. The class contains several members: a field named `top`, two methods named `Push` and `Pop`, and a nested class named `Entry`. The `Entry` class further contains three members: a field named `next`, a field named `data`, and a constructor. Assuming that the source code of the example is stored in the file `acme.cs`, the command line

```
csc /t:library acme.cs
```

compiles the example as a library (code without a `Main` entry point) and produces an assembly named `acme.dll`.

Assemblies contain executable code in the form of *Intermediate Language* (IL) instructions, and symbolic information in the form of *metadata*. Before it is executed, the IL code in an assembly is automatically converted to processor-specific code by the Just-In-Time (JIT) compiler of .NET Common Language Runtime.

Because an assembly is a self-describing unit of functionality containing both code and metadata, there is no need for `#include` directives and header files in C#. The public types and members contained in a particular assembly are made available in a C# program simply by referencing that assembly when compiling the program. For example, this program uses the `Acme.Collections.Stack` class from the `acme.dll` assembly:

```

using System;
using Acme.Collections;

class Test
{
    static void Main() {
        Stack s = new Stack();
        s.Push(1);
        s.Push(10);
        s.Push(100);
        Console.WriteLine(s.Pop());
        Console.WriteLine(s.Pop());
        Console.WriteLine(s.Pop());
    }
}

```

If the program is stored in the file `test.cs`, when `test.cs` is compiled, the `acme.dll` assembly can be referenced using the compiler's `/r` option:

```
csc /r:acme.dll test.cs
```

This creates an executable assembly named `test.exe`, which, when run, produces the output:

```

100
10
1

```

C# permits the source text of a program to be stored in several source files. When a multi-file C# program is compiled, all of the source files are processed together, and the source files can freely reference each other—conceptually, it is as if all the source files were concatenated into one large file before being processed. Forward declarations are never needed in C# because, with very few exceptions, declaration order is insignificant. C# does not limit a source file to declaring only one public type nor does it require the name of the source file to match a type declared in the source file.

Types and variables

There are two kinds of types in C#: *value types* and *reference types*. Variables of value types directly contain their data whereas variables of reference types store references to their data, the latter being known as objects. With reference types, it is possible for two variables to reference the same object and thus possible for operations on one variable to affect the object referenced by the other variable. With value types, the variables each have their own copy of the data, and it is not possible for operations on one to affect the other (except in the case of `ref` and `out` parameter variables).

C#'s value types are further divided into *simple types*, *enum types*, *struct types*, and *nullable types*, and C#'s reference types are further divided into *class types*, *interface types*, *array types*, and *delegate types*.

The following table provides an overview of C#'s type system.

CATEGORY	TYPES	DESCRIPTION
Value types	Simple types	Signed integral: <code>sbyte</code> , <code>short</code> , <code>int</code> , <code>long</code>
		Unsigned integral: <code>byte</code> , <code>ushort</code> , <code>uint</code> , <code>ulong</code>

CATEGORY	TYPES	DESCRIPTION
		Unicode characters: <code>char</code>
		IEEE floating point: <code>float</code> , <code>double</code>
		High-precision decimal: <code>decimal</code>
		Boolean: <code>bool</code>
	Enum types	User-defined types of the form <code>enum E {...}</code>
	Struct types	User-defined types of the form <code>struct S {...}</code>
	Nullable types	Extensions of all other value types with a <code>null</code> value
Reference types	Class types	Ultimate base class of all other types: <code>object</code>
		Unicode strings: <code>string</code>
		User-defined types of the form <code>class C {...}</code>
	Interface types	User-defined types of the form <code>interface I {...}</code>
	Array types	Single- and multi-dimensional, for example, <code>int[]</code> and <code>int[,]</code>
	Delegate types	User-defined types of the form e.g. <code>delegate int D(...)</code>

The eight integral types provide support for 8-bit, 16-bit, 32-bit, and 64-bit values in signed or unsigned form.

The two floating point types, `float` and `double`, are represented using the 32-bit single-precision and 64-bit double-precision IEEE 754 formats.

The `decimal` type is a 128-bit data type suitable for financial and monetary calculations.

C#'s `bool` type is used to represent boolean values—values that are either `true` or `false`.

Character and string processing in C# uses Unicode encoding. The `char` type represents a UTF-16 code unit, and the `string` type represents a sequence of UTF-16 code units.

The following table summarizes C#'s numeric types.

CATEGORY	BITS	TYPE	RANGE/PRECISION
Signed integral	8	<code>sbyte</code>	-128...127

CATEGORY	BITS	TYPE	RANGE/PRECISION
	16	short	-32,768...32,767
	32	int	-2,147,483,648...2,147,483,647
	64	long	-9,223,372,036,854,775,808...9,223,372,036,854,775,807
Unsigned integral	8	byte	0...255
	16	ushort	0...65,535
	32	uint	0...4,294,967,295
	64	ulong	0...18,446,744,073,709,551,615
Floating point	32	float	1.5×10^{-45} to 3.4×10^{38} , 7-digit precision
	64	double	5.0×10^{-324} to 1.7×10^{308} , 15-digit precision
Decimal	128	decimal	1.0×10^{-28} to 7.9×10^{28} , 28-digit precision

C# programs use ***type declarations*** to create new types. A type declaration specifies the name and the members of the new type. Five of C#'s categories of types are user-definable: class types, struct types, interface types, enum types, and delegate types.

A class type defines a data structure that contains data members (fields) and function members (methods, properties, and others). Class types support single inheritance and polymorphism, mechanisms whereby derived classes can extend and specialize base classes.

A struct type is similar to a class type in that it represents a structure with data members and function members. However, unlike classes, structs are value types and do not require heap allocation. Struct types do not support user-specified inheritance, and all struct types implicitly inherit from type `object`.

An interface type defines a contract as a named set of public function members. A class or struct that implements an interface must provide implementations of the interface's function members. An interface may inherit from multiple base interfaces, and a class or struct may implement multiple interfaces.

A delegate type represents references to methods with a particular parameter list and return type. Delegates make it possible to treat methods as entities that can be assigned to variables and passed as parameters. Delegates are similar to the concept of function pointers found in some other languages, but unlike function pointers, delegates are object-oriented and type-safe.

Class, struct, interface and delegate types all support generics, whereby they can be parameterized with other

types.

An enum type is a distinct type with named constants. Every enum type has an underlying type, which must be one of the eight integral types. The set of values of an enum type is the same as the set of values of the underlying type.

C# supports single- and multi-dimensional arrays of any type. Unlike the types listed above, array types do not have to be declared before they can be used. Instead, array types are constructed by following a type name with square brackets. For example, `int[]` is a single-dimensional array of `int`, `int[,]` is a two-dimensional array of `int`, and `int[][]` is a single-dimensional array of single-dimensional arrays of `int`.

Nullable types also do not have to be declared before they can be used. For each non-nullable value type `T` there is a corresponding nullable type `T?`, which can hold an additional value `null`. For instance, `int?` is a type that can hold any 32 bit integer or the value `null`.

C#'s type system is unified such that a value of any type can be treated as an object. Every type in C# directly or indirectly derives from the `object` class type, and `object` is the ultimate base class of all types. Values of reference types are treated as objects simply by viewing the values as type `object`. Values of value types are treated as objects by performing **boxing** and **unboxing** operations. In the following example, an `int` value is converted to `object` and back again to `int`.

```
using System;

class Test
{
    static void Main() {
        int i = 123;
        object o = i;           // Boxing
        int j = (int)o;         // Unboxing
    }
}
```

When a value of a value type is converted to type `object`, an object instance, also called a "box," is allocated to hold the value, and the value is copied into that box. Conversely, when an `object` reference is cast to a value type, a check is made that the referenced object is a box of the correct value type, and, if the check succeeds, the value in the box is copied out.

C#'s unified type system effectively means that value types can become objects "on demand." Because of the unification, general-purpose libraries that use type `object` can be used with both reference types and value types.

There are several kinds of **variables** in C#, including fields, array elements, local variables, and parameters. Variables represent storage locations, and every variable has a type that determines what values can be stored in the variable, as shown by the following table.

TYPE OF VARIABLE	POSSIBLE CONTENTS
Non-nullable value type	A value of that exact type
Nullable value type	A null value or a value of that exact type
<code>object</code>	A null reference, a reference to an object of any reference type, or a reference to a boxed value of any value type

TYPE OF VARIABLE	POSSIBLE CONTENTS
Class type	A null reference, a reference to an instance of that class type, or a reference to an instance of a class derived from that class type
Interface type	A null reference, a reference to an instance of a class type that implements that interface type, or a reference to a boxed value of a value type that implements that interface type
Array type	A null reference, a reference to an instance of that array type, or a reference to an instance of a compatible array type
Delegate type	A null reference or a reference to an instance of that delegate type

Expressions

Expressions are constructed from **operands** and **operators**. The operators of an expression indicate which operations to apply to the operands. Examples of operators include `+`, `-`, `*`, `/`, and `new`. Examples of operands include literals, fields, local variables, and expressions.

When an expression contains multiple operators, the **precedence** of the operators controls the order in which the individual operators are evaluated. For example, the expression `x + y * z` is evaluated as `x + (y * z)` because the `*` operator has higher precedence than the `+` operator.

Most operators can be **overloaded**. Operator overloading permits user-defined operator implementations to be specified for operations where one or both of the operands are of a user-defined class or struct type.

The following table summarizes C#'s operators, listing the operator categories in order of precedence from highest to lowest. Operators in the same category have equal precedence.

CATEGORY	EXPRESSION	DESCRIPTION
Primary	<code>x.m</code>	Member access
	<code>x(...)</code>	Method and delegate invocation
	<code>x[...]</code>	Array and indexer access
	<code>x++</code>	Post-increment
	<code>x--</code>	Post-decrement
	<code>new T(...)</code>	Object and delegate creation
	<code>new T(...){...}</code>	Object creation with initializer
	<code>new {...}</code>	Anonymous object initializer
	<code>new T[...]</code>	Array creation

CATEGORY	EXPRESSION	DESCRIPTION
	<code>typeof(T)</code>	Obtain <code>System.Type</code> object for <code>T</code>
	<code>checked(x)</code>	Evaluate expression in checked context
	<code>unchecked(x)</code>	Evaluate expression in unchecked context
	<code>default(T)</code>	Obtain default value of type <code>T</code>
	<code>delegate {...}</code>	Anonymous function (anonymous method)
Unary	<code>+x</code>	Identity
	<code>-x</code>	Negation
	<code>!x</code>	Logical negation
	<code>~x</code>	Bitwise negation
	<code>++x</code>	Pre-increment
	<code>--x</code>	Pre-decrement
	<code>(T)x</code>	Explicitly convert <code>x</code> to type <code>T</code>
	<code>await x</code>	Asynchronously wait for <code>x</code> to complete
Multiplicative	<code>x * y</code>	Multiplication
	<code>x / y</code>	Division
	<code>x % y</code>	Remainder
Additive	<code>x + y</code>	Addition, string concatenation, delegate combination
	<code>x - y</code>	Subtraction, delegate removal
Shift	<code>x << y</code>	Shift left
	<code>x >> y</code>	Shift right
Relational and type testing	<code>x < y</code>	Less than
	<code>x > y</code>	Greater than
	<code>x <= y</code>	Less than or equal

CATEGORY	EXPRESSION	DESCRIPTION
	<code>x >= y</code>	Greater than or equal
	<code>x is T</code>	Return <code>true</code> if <code>x</code> is a <code>T</code> , <code>false</code> otherwise
	<code>x as T</code>	Return <code>x</code> typed as <code>T</code> , or <code>null</code> if <code>x</code> is not a <code>T</code>
Equality	<code>x == y</code>	Equal
	<code>x != y</code>	Not equal
Logical AND	<code>x & y</code>	Integer bitwise AND, boolean logical AND
Logical XOR	<code>x ^ y</code>	Integer bitwise XOR, boolean logical XOR
Logical OR	<code>x y</code>	Integer bitwise OR, boolean logical OR
Conditional AND	<code>x && y</code>	Evaluates <code>y</code> only if <code>x</code> is <code>true</code>
Conditional OR	<code>x y</code>	Evaluates <code>y</code> only if <code>x</code> is <code>false</code>
Null coalescing	<code>x ?? y</code>	Evaluates to <code>y</code> if <code>x</code> is <code>null</code> , to <code>x</code> otherwise
Conditional	<code>x ? y : z</code>	Evaluates <code>y</code> if <code>x</code> is <code>true</code> , <code>z</code> if <code>x</code> is <code>false</code>
Assignment or anonymous function	<code>x = y</code>	Assignment
	<code>x op= y</code>	Compound assignment; supported operators are <code>*=</code> , <code>/=</code> , <code>%=</code> , <code>+=</code> , <code>-=</code> , <code><<=</code> , <code>>>=</code> , <code>&=</code> , <code>^=</code> , <code> =</code>
	<code>(T x) => y</code>	Anonymous function (lambda expression)

Statements

The actions of a program are expressed using **statements**. C# supports several different kinds of statements, a number of which are defined in terms of embedded statements.

A **block** permits multiple statements to be written in contexts where a single statement is allowed. A block consists of a list of statements written between the delimiters `{` and `}`.

Declaration statements are used to declare local variables and constants.

Expression statements are used to evaluate expressions. Expressions that can be used as statements include method invocations, object allocations using the `new` operator, assignments using `=` and the compound assignment operators, increment and decrement operations using the `++` and `--` operators and `await`

expressions.

Selection statements are used to select one of a number of possible statements for execution based on the value of some expression. In this group are the `if` and `switch` statements.

Iteration statements are used to repeatedly execute an embedded statement. In this group are the `while`, `do`, `for`, and `foreach` statements.

Jump statements are used to transfer control. In this group are the `break`, `continue`, `goto`, `throw`, `return`, and `yield` statements.

The `try ... catch` statement is used to catch exceptions that occur during execution of a block, and the `try ... finally` statement is used to specify finalization code that is always executed, whether an exception occurred or not.

The `checked` and `unchecked` statements are used to control the overflow checking context for integral-type arithmetic operations and conversions.

The `lock` statement is used to obtain the mutual-exclusion lock for a given object, execute a statement, and then release the lock.

The `using` statement is used to obtain a resource, execute a statement, and then dispose of that resource.

Below are examples of each kind of statement

Local variable declarations

```
static void Main() {  
    int a;  
    int b = 2, c = 3;  
    a = 1;  
    Console.WriteLine(a + b + c);  
}
```

Local constant declaration

```
static void Main() {  
    const float pi = 3.1415927f;  
    const int r = 25;  
    Console.WriteLine(pi * r * r);  
}
```

Expression statement

```
static void Main() {  
    int i;  
    i = 123;           // Expression statement  
    Console.WriteLine(i); // Expression statement  
    i++;              // Expression statement  
    Console.WriteLine(i); // Expression statement  
}
```

`if` statement

```
static void Main(string[] args) {
    if (args.Length == 0) {
        Console.WriteLine("No arguments");
    }
    else {
        Console.WriteLine("One or more arguments");
    }
}
```

switch statement

```
static void Main(string[] args) {
    int n = args.Length;
    switch (n) {
        case 0:
            Console.WriteLine("No arguments");
            break;
        case 1:
            Console.WriteLine("One argument");
            break;
        default:
            Console.WriteLine("{0} arguments", n);
            break;
    }
}
```

while statement

```
static void Main(string[] args) {
    int i = 0;
    while (i < args.Length) {
        Console.WriteLine(args[i]);
        i++;
    }
}
```

do statement

```
static void Main() {
    string s;
    do {
        s = Console.ReadLine();
        if (s != null) Console.WriteLine(s);
    } while (s != null);
}
```

for statement

```
static void Main(string[] args) {
    for (int i = 0; i < args.Length; i++) {
        Console.WriteLine(args[i]);
    }
}
```

foreach statement


```
static void Main(string[] args) {  
    foreach (string s in args) {  
        Console.WriteLine(s);  
    }  
}
```

break statement

```
static void Main() {  
    while (true) {  
        string s = Console.ReadLine();  
        if (s == null) break;  
        Console.WriteLine(s);  
    }  
}
```

continue statement

```
static void Main(string[] args) {  
    for (int i = 0; i < args.Length; i++) {  
        if (args[i].StartsWith("/")) continue;  
        Console.WriteLine(args[i]);  
    }  
}
```

goto statement

```
static void Main(string[] args) {  
    int i = 0;  
    goto check;  
loop:  
    Console.WriteLine(args[i++]);  
check:  
    if (i < args.Length) goto loop;  
}
```

return statement

```
static int Add(int a, int b) {  
    return a + b;  
}  
  
static void Main() {  
    Console.WriteLine(Add(1, 2));  
    return;  
}
```

yield statement

```

static IEnumerable<int> Range(int from, int to) {
    for (int i = from; i < to; i++) {
        yield return i;
    }
    yield break;
}

static void Main() {
    foreach (int x in Range(-10,10)) {
        Console.WriteLine(x);
    }
}

```

throw and **try** statements

```

static double Divide(double x, double y) {
    if (y == 0) throw new DivideByZeroException();
    return x / y;
}

static void Main(string[] args) {
    try {
        if (args.Length != 2) {
            throw new Exception("Two numbers required");
        }
        double x = double.Parse(args[0]);
        double y = double.Parse(args[1]);
        Console.WriteLine(Divide(x, y));
    }
    catch (Exception e) {
        Console.WriteLine(e.Message);
    }
    finally {
        Console.WriteLine("Good bye!");
    }
}

```

checked and **unchecked** statements

```

static void Main() {
    int i = int.MaxValue;
    checked {
        Console.WriteLine(i + 1);    // Exception
    }
    unchecked {
        Console.WriteLine(i + 1);    // Overflow
    }
}

```

lock statement

```

class Account
{
    decimal balance;
    public void Withdraw(decimal amount) {
        lock (this) {
            if (amount > balance) {
                throw new Exception("Insufficient funds");
            }
            balance -= amount;
        }
    }
}

```

using **statement**

```

static void Main() {
    using (TextWriter w = File.CreateText("test.txt")) {
        w.WriteLine("Line one");
        w.WriteLine("Line two");
        w.WriteLine("Line three");
    }
}

```

Classes and objects

Classes are the most fundamental of C#'s types. A class is a data structure that combines state (fields) and actions (methods and other function members) in a single unit. A class provides a definition for dynamically created *instances* of the class, also known as *objects*. Classes support *inheritance* and *polymorphism*, mechanisms whereby *derived classes* can extend and specialize *base classes*.

New classes are created using class declarations. A class declaration starts with a header that specifies the attributes and modifiers of the class, the name of the class, the base class (if given), and the interfaces implemented by the class. The header is followed by the class body, which consists of a list of member declarations written between the delimiters `{` and `}`.

The following is a declaration of a simple class named `Point`:

```

public class Point
{
    public int x, y;

    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }
}

```

Instances of classes are created using the `new` operator, which allocates memory for a new instance, invokes a constructor to initialize the instance, and returns a reference to the instance. The following statements create two `Point` objects and store references to those objects in two variables:

```

Point p1 = new Point(0, 0);
Point p2 = new Point(10, 20);

```

The memory occupied by an object is automatically reclaimed when the object is no longer in use. It is neither necessary nor possible to explicitly deallocate objects in C#.

Members

The members of a class are either *static members* or *instance members*. Static members belong to classes, and instance members belong to objects (instances of classes).

The following table provides an overview of the kinds of members a class can contain.

MEMBER	DESCRIPTION
Constants	Constant values associated with the class
Fields	Variables of the class
Methods	Computations and actions that can be performed by the class
Properties	Actions associated with reading and writing named properties of the class
Indexers	Actions associated with indexing instances of the class like an array
Events	Notifications that can be generated by the class
Operators	Conversions and expression operators supported by the class
Constructors	Actions required to initialize instances of the class or the class itself
Destructors	Actions to perform before instances of the class are permanently discarded
Types	Nested types declared by the class

Accessibility

Each member of a class has an associated accessibility, which controls the regions of program text that are able to access the member. There are five possible forms of accessibility. These are summarized in the following table.

ACCESSIBILITY	MEANING
<code>public</code>	Access not limited
<code>protected</code>	Access limited to this class or classes derived from this class
<code>internal</code>	Access limited to this program
<code>protected internal</code>	Access limited to this program or classes derived from this class
<code>private</code>	Access limited to this class

Type parameters

A class definition may specify a set of type parameters by following the class name with angle brackets enclosing a list of type parameter names. The type parameters can then be used in the body of the class

declarations to define the members of the class. In the following example, the type parameters of `Pair` are `TFirst` and `TSecond` :

```
public class Pair<TFirst,TSecond>
{
    public TFirst First;
    public TSecond Second;
}
```

A class type that is declared to take type parameters is called a generic class type. Struct, interface and delegate types can also be generic.

When the generic class is used, type arguments must be provided for each of the type parameters:

```
Pair<int,string> pair = new Pair<int,string> { First = 1, Second = "two" };
int i = pair.First;    // TFirst is int
string s = pair.Second; // TSecond is string
```

A generic type with type arguments provided, like `Pair<int,string>` above, is called a constructed type.

Base classes

A class declaration may specify a base class by following the class name and type parameters with a colon and the name of the base class. Omitting a base class specification is the same as deriving from type `object` . In the following example, the base class of `Point3D` is `Point` , and the base class of `Point` is `object` :

```
public class Point
{
    public int x, y;

    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }
}

public class Point3D: Point
{
    public int z;

    public Point3D(int x, int y, int z): base(x, y) {
        this.z = z;
    }
}
```

A class inherits the members of its base class. Inheritance means that a class implicitly contains all members of its base class, except for the instance and static constructors, and the destructors of the base class. A derived class can add new members to those it inherits, but it cannot remove the definition of an inherited member. In the previous example, `Point3D` inherits the `x` and `y` fields from `Point` , and every `Point3D` instance contains three fields, `x` , `y` , and `z` .

An implicit conversion exists from a class type to any of its base class types. Therefore, a variable of a class type can reference an instance of that class or an instance of any derived class. For example, given the previous class declarations, a variable of type `Point` can reference either a `Point` or a `Point3D` :

```
Point a = new Point(10, 20);
Point b = new Point3D(10, 20, 30);
```

Fields

A field is a variable that is associated with a class or with an instance of a class.

A field declared with the `static` modifier defines a **static field**. A static field identifies exactly one storage location. No matter how many instances of a class are created, there is only ever one copy of a static field.

A field declared without the `static` modifier defines an **instance field**. Every instance of a class contains a separate copy of all the instance fields of that class.

In the following example, each instance of the `Color` class has a separate copy of the `r`, `g`, and `b` instance fields, but there is only one copy of the `Black`, `White`, `Red`, `Green`, and `Blue` static fields:

```
public class Color
{
    public static readonly Color Black = new Color(0, 0, 0);
    public static readonly Color White = new Color(255, 255, 255);
    public static readonly Color Red = new Color(255, 0, 0);
    public static readonly Color Green = new Color(0, 255, 0);
    public static readonly Color Blue = new Color(0, 0, 255);
    private byte r, g, b;

    public Color(byte r, byte g, byte b) {
        this.r = r;
        this.g = g;
        this.b = b;
    }
}
```

As shown in the previous example, **read-only fields** may be declared with a `readonly` modifier. Assignment to a `readonly` field can only occur as part of the field's declaration or in a constructor in the same class.

Methods

A **method** is a member that implements a computation or action that can be performed by an object or class.

Static methods are accessed through the class. **Instance methods** are accessed through instances of the class.

Methods have a (possibly empty) list of **parameters**, which represent values or variable references passed to the method, and a **return type**, which specifies the type of the value computed and returned by the method. A method's return type is `void` if it does not return a value.

Like types, methods may also have a set of type parameters, for which type arguments must be specified when the method is called. Unlike types, the type arguments can often be inferred from the arguments of a method call and need not be explicitly given.

The **signature** of a method must be unique in the class in which the method is declared. The signature of a method consists of the name of the method, the number of type parameters and the number, modifiers, and types of its parameters. The signature of a method does not include the return type.

Parameters

Parameters are used to pass values or variable references to methods. The parameters of a method get their actual values from the **arguments** that are specified when the method is invoked. There are four kinds of parameters: value parameters, reference parameters, output parameters, and parameter arrays.

A **value parameter** is used for input parameter passing. A value parameter corresponds to a local variable that gets its initial value from the argument that was passed for the parameter. Modifications to a value parameter do not affect the argument that was passed for the parameter.

Value parameters can be optional, by specifying a default value so that corresponding arguments can be omitted.

A **reference parameter** is used for both input and output parameter passing. The argument passed for a reference parameter must be a variable, and during execution of the method, the reference parameter represents the same storage location as the argument variable. A reference parameter is declared with the `ref` modifier. The following example shows the use of `ref` parameters.

```
using System;

class Test
{
    static void Swap(ref int x, ref int y) {
        int temp = x;
        x = y;
        y = temp;
    }

    static void Main() {
        int i = 1, j = 2;
        Swap(ref i, ref j);
        Console.WriteLine("{0} {1}", i, j);           // Outputs "2 1"
    }
}
```

An **output parameter** is used for output parameter passing. An output parameter is similar to a reference parameter except that the initial value of the caller-provided argument is unimportant. An output parameter is declared with the `out` modifier. The following example shows the use of `out` parameters.

```
using System;

class Test
{
    static void Divide(int x, int y, out int result, out int remainder) {
        result = x / y;
        remainder = x % y;
    }

    static void Main() {
        int res, rem;
        Divide(10, 3, out res, out rem);
        Console.WriteLine("{0} {1}", res, rem);      // Outputs "3 1"
    }
}
```

A **parameter array** permits a variable number of arguments to be passed to a method. A parameter array is declared with the `params` modifier. Only the last parameter of a method can be a parameter array, and the type of a parameter array must be a single-dimensional array type. The `Write` and `WriteLine` methods of the `System.Console` class are good examples of parameter array usage. They are declared as follows.

```
public class Console
{
    public static void Write(string fmt, params object[] args) {...}
    public static void WriteLine(string fmt, params object[] args) {...}
    ...
}
```

Within a method that uses a parameter array, the parameter array behaves exactly like a regular parameter of an array type. However, in an invocation of a method with a parameter array, it is possible to pass either a single argument of the parameter array type or any number of arguments of the element type of the parameter array. In the latter case, an array instance is automatically created and initialized with the given arguments. This example

```
Console.WriteLine("x={0} y={1} z={2}", x, y, z);
```

is equivalent to writing the following.

```
string s = "x={0} y={1} z={2}";
object[] args = new object[3];
args[0] = x;
args[1] = y;
args[2] = z;
Console.WriteLine(s, args);
```

Method body and local variables

A method's body specifies the statements to execute when the method is invoked.

A method body can declare variables that are specific to the invocation of the method. Such variables are called **local variables**. A local variable declaration specifies a type name, a variable name, and possibly an initial value. The following example declares a local variable `i` with an initial value of zero and a local variable `j` with no initial value.

```
using System;

class Squares
{
    static void Main() {
        int i = 0;
        int j;
        while (i < 10) {
            j = i * i;
            Console.WriteLine("{0} x {0} = {1}", i, j);
            i = i + 1;
        }
    }
}
```

C# requires a local variable to be **definitely assigned** before its value can be obtained. For example, if the declaration of the previous `i` did not include an initial value, the compiler would report an error for the subsequent usages of `i` because `i` would not be definitely assigned at those points in the program.

A method can use `return` statements to return control to its caller. In a method returning `void`, `return` statements cannot specify an expression. In a method returning non-`void`, `return` statements must include an expression that computes the return value.

Static and instance methods

A method declared with a `static` modifier is a **static method**. A static method does not operate on a specific instance and can only directly access static members.

A method declared without a `static` modifier is an **instance method**. An instance method operates on a specific instance and can access both static and instance members. The instance on which an instance method was invoked can be explicitly accessed as `this`. It is an error to refer to `this` in a static method.

The following `Entity` class has both static and instance members.


```

class Entity
{
    static int nextSerialNo;
    int serialNo;

    public Entity() {
        serialNo = nextSerialNo++;
    }

    public int GetSerialNo() {
        return serialNo;
    }

    public static int GetNextSerialNo() {
        return nextSerialNo;
    }

    public static void SetNextSerialNo(int value) {
        nextSerialNo = value;
    }
}

```

Each `Entity` instance contains a serial number (and presumably some other information that is not shown here). The `Entity` constructor (which is like an instance method) initializes the new instance with the next available serial number. Because the constructor is an instance member, it is permitted to access both the `serialNo` instance field and the `nextSerialNo` static field.

The `GetNextSerialNo` and `SetNextSerialNo` static methods can access the `nextSerialNo` static field, but it would be an error for them to directly access the `serialNo` instance field.

The following example shows the use of the `Entity` class.

```

using System;

class Test
{
    static void Main() {
        Entity.SetNextSerialNo(1000);
        Entity e1 = new Entity();
        Entity e2 = new Entity();
        Console.WriteLine(e1.GetSerialNo());           // Outputs "1000"
        Console.WriteLine(e2.GetSerialNo());           // Outputs "1001"
        Console.WriteLine(Entity.GetNextSerialNo());   // Outputs "1002"
    }
}

```

Note that the `SetNextSerialNo` and `GetNextSerialNo` static methods are invoked on the class whereas the `GetSerialNo` instance method is invoked on instances of the class.

Virtual, override, and abstract methods

When an instance method declaration includes a `virtual` modifier, the method is said to be a *virtual method*. When no `virtual` modifier is present, the method is said to be a *non-virtual method*.

When a virtual method is invoked, the *run-time type* of the instance for which that invocation takes place determines the actual method implementation to invoke. In a nonvirtual method invocation, the *compile-time type* of the instance is the determining factor.

A virtual method can be *overridden* in a derived class. When an instance method declaration includes an `override` modifier, the method overrides an inherited virtual method with the same signature. Whereas a virtual method declaration introduces a new method, an override method declaration specializes an existing

inherited virtual method by providing a new implementation of that method.

An ***abstract*** method is a virtual method with no implementation. An abstract method is declared with the `abstract` modifier and is permitted only in a class that is also declared `abstract`. An abstract method must be overridden in every non-abstract derived class.

The following example declares an abstract class, `Expression`, which represents an expression tree node, and three derived classes, `Constant`, `VariableReference`, and `Operation`, which implement expression tree nodes for constants, variable references, and arithmetic operations. (This is similar to, but not to be confused with the expression tree types introduced in [Expression tree types](#)).

```

using System;
using System.Collections;

public abstract class Expression
{
    public abstract double Evaluate(Hashtable vars);
}

public class Constant: Expression
{
    double value;

    public Constant(double value) {
        this.value = value;
    }

    public override double Evaluate(Hashtable vars) {
        return value;
    }
}

public class VariableReference: Expression
{
    string name;

    public VariableReference(string name) {
        this.name = name;
    }

    public override double Evaluate(Hashtable vars) {
        object value = vars[name];
        if (value == null) {
            throw new Exception("Unknown variable: " + name);
        }
        return Convert.ToDouble(value);
    }
}

public class Operation: Expression
{
    Expression left;
    char op;
    Expression right;

    public Operation(Expression left, char op, Expression right) {
        this.left = left;
        this.op = op;
        this.right = right;
    }

    public override double Evaluate(Hashtable vars) {
        double x = left.Evaluate(vars);
        double y = right.Evaluate(vars);
        switch (op) {
            case '+': return x + y;
            case '-': return x - y;
            case '*': return x * y;
            case '/': return x / y;
        }
        throw new Exception("Unknown operator");
    }
}

```

The previous four classes can be used to model arithmetic expressions. For example, using instances of these classes, the expression `x + 3` can be represented as follows.

```
Expression e = new Operation(
    new VariableReference("x"),
    '+',
    new Constant(3));
```

The `Evaluate` method of an `Expression` instance is invoked to evaluate the given expression and produce a `double` value. The method takes as an argument a `Hashtable` that contains variable names (as keys of the entries) and values (as values of the entries). The `Evaluate` method is a virtual abstract method, meaning that non-abstract derived classes must override it to provide an actual implementation.

A `Constant`'s implementation of `Evaluate` simply returns the stored constant. A `VariableReference`'s implementation looks up the variable name in the hashtable and returns the resulting value. An `Operation`'s implementation first evaluates the left and right operands (by recursively invoking their `Evaluate` methods) and then performs the given arithmetic operation.

The following program uses the `Expression` classes to evaluate the expression `x * (y + 2)` for different values of `x` and `y`.

```
using System;
using System.Collections;

class Test
{
    static void Main() {
        Expression e = new Operation(
            new VariableReference("x"),
            '*',
            new Operation(
                new VariableReference("y"),
                '+',
                new Constant(2)
            )
        );
        Hashtable vars = new Hashtable();
        vars["x"] = 3;
        vars["y"] = 5;
        Console.WriteLine(e.Evaluate(vars));           // Outputs "21"
        vars["x"] = 1.5;
        vars["y"] = 9;
        Console.WriteLine(e.Evaluate(vars));           // Outputs "16.5"
    }
}
```

Method overloading

Method **overloading** permits multiple methods in the same class to have the same name as long as they have unique signatures. When compiling an invocation of an overloaded method, the compiler uses **overload resolution** to determine the specific method to invoke. Overload resolution finds the one method that best matches the arguments or reports an error if no single best match can be found. The following example shows overload resolution in effect. The comment for each invocation in the `Main` method shows which method is actually invoked.

```

class Test
{
    static void F() {
        Console.WriteLine("F()");
    }

    static void F(object x) {
        Console.WriteLine("F(object)");
    }

    static void F(int x) {
        Console.WriteLine("F(int)");
    }

    static void F(double x) {
        Console.WriteLine("F(double)");
    }

    static void F<T>(T x) {
        Console.WriteLine("F<T>(T)");
    }

    static void F(double x, double y) {
        Console.WriteLine("F(double, double)");
    }

    static void Main() {
        F();           // Invokes F()
        F(1);          // Invokes F(int)
        F(1.0);        // Invokes F(double)
        F("abc");      // Invokes F(object)
        F((double)1);   // Invokes F(double)
        F((object)1);   // Invokes F(object)
        F<int>(1);      // Invokes F<T>(T)
        F(1, 1);       // Invokes F(double, double)
    }
}

```

As shown by the example, a particular method can always be selected by explicitly casting the arguments to the exact parameter types and/or explicitly supplying type arguments.

Other function members

Members that contain executable code are collectively known as the *function members* of a class. The preceding section describes methods, which are the primary kind of function members. This section describes the other kinds of function members supported by C#: constructors, properties, indexers, events, operators, and destructors.

The following code shows a generic class called `List<T>`, which implements a growable list of objects. The class contains several examples of the most common kinds of function members.

```

public class List<T> {
    // Constant...
    const int defaultCapacity = 4;

    // Fields...
    T[] items;
    int count;

    // Constructors...
    public List(int capacity = defaultCapacity) {
        items = new T[capacity];
    }

    // Properties...

```

```

// ...
public int Count {
    get { return count; }
}
public int Capacity {
    get {
        return items.Length;
    }
    set {
        if (value < count) value = count;
        if (value != items.Length) {
            T[] newItems = new T[value];
            Array.Copy(items, 0, newItems, 0, count);
            items = newItems;
        }
    }
}

// Indexer...
public T this[int index] {
    get {
        return items[index];
    }
    set {
        items[index] = value;
        OnChanged();
    }
}

// Methods...
public void Add(T item) {
    if (count == Capacity) Capacity = count * 2;
    items[count] = item;
    count++;
    OnChanged();
}
protected virtual void OnChanged() {
    if (Changed != null) Changed(this, EventArgs.Empty);
}
public override bool Equals(object other) {
    return Equals(this, other as List<T>);
}
static bool Equals(List<T> a, List<T> b) {
    if (a == null) return b == null;
    if (b == null || a.count != b.count) return false;
    for (int i = 0; i < a.count; i++) {
        if (!object.Equals(a.items[i], b.items[i])) {
            return false;
        }
    }
    return true;
}

// Event...
public event EventHandler Changed;

// Operators...
public static bool operator ==(List<T> a, List<T> b) {
    return Equals(a, b);
}
public static bool operator !=(List<T> a, List<T> b) {
    return !Equals(a, b);
}
}

```

Constructors

C# supports both instance and static constructors. An *instance constructor* is a member that implements the actions required to initialize an instance of a class. A *static constructor* is a member that implements the

actions required to initialize a class itself when it is first loaded.

A constructor is declared like a method with no return type and the same name as the containing class. If a constructor declaration includes a `static` modifier, it declares a static constructor. Otherwise, it declares an instance constructor.

Instance constructors can be overloaded. For example, the `List<T>` class declares two instance constructors, one with no parameters and one that takes an `int` parameter. Instance constructors are invoked using the `new` operator. The following statements allocate two `List<string>` instances using each of the constructors of the `List` class.

```
List<string> list1 = new List<string>();  
List<string> list2 = new List<string>(10);
```

Unlike other members, instance constructors are not inherited, and a class has no instance constructors other than those actually declared in the class. If no instance constructor is supplied for a class, then an empty one with no parameters is automatically provided.

Properties

Properties are a natural extension of fields. Both are named members with associated types, and the syntax for accessing fields and properties is the same. However, unlike fields, properties do not denote storage locations. Instead, properties have **accessors** that specify the statements to be executed when their values are read or written.

A property is declared like a field, except that the declaration ends with a `get` accessor and/or a `set` accessor written between the delimiters `{` and `}` instead of ending in a semicolon. A property that has both a `get` accessor and a `set` accessor is a **read-write property**, a property that has only a `get` accessor is a **read-only property**, and a property that has only a `set` accessor is a **write-only property**.

A `get` accessor corresponds to a parameterless method with a return value of the property type. Except as the target of an assignment, when a property is referenced in an expression, the `get` accessor of the property is invoked to compute the value of the property.

A `set` accessor corresponds to a method with a single parameter named `value` and no return type. When a property is referenced as the target of an assignment or as the operand of `++` or `--`, the `set` accessor is invoked with an argument that provides the new value.

The `List<T>` class declares two properties, `Count` and `Capacity`, which are read-only and read-write, respectively. The following is an example of use of these properties.

```
List<string> names = new List<string>();  
names.Capacity = 100;           // Invokes set accessor  
int i = names.Count;           // Invokes get accessor  
int j = names.Capacity;        // Invokes get accessor
```

Similar to fields and methods, C# supports both instance properties and static properties. Static properties are declared with the `static` modifier, and instance properties are declared without it.

The accessor(s) of a property can be virtual. When a property declaration includes a `virtual`, `abstract`, or `override` modifier, it applies to the accessor(s) of the property.

Indexers

An **indexer** is a member that enables objects to be indexed in the same way as an array. An indexer is declared like a property except that the name of the member is `this` followed by a parameter list written between the delimiters `[` and `]`. The parameters are available in the accessor(s) of the indexer. Similar to properties, indexers can be read-write, read-only, and write-only, and the accessor(s) of an indexer can be virtual.

The `List` class declares a single read-write indexer that takes an `int` parameter. The indexer makes it possible to index `List` instances with `int` values. For example

```
List<string> names = new List<string>();
names.Add("Liz");
names.Add("Martha");
names.Add("Beth");
for (int i = 0; i < names.Count; i++) {
    string s = names[i];
    names[i] = s.ToUpper();
}
```

Indexers can be overloaded, meaning that a class can declare multiple indexers as long as the number or types of their parameters differ.

Events

An **event** is a member that enables a class or object to provide notifications. An event is declared like a field except that the declaration includes an `event` keyword and the type must be a delegate type.

Within a class that declares an event member, the event behaves just like a field of a delegate type (provided the event is not abstract and does not declare accessors). The field stores a reference to a delegate that represents the event handlers that have been added to the event. If no event handles are present, the field is `null`.

The `List<T>` class declares a single event member called `Changed`, which indicates that a new item has been added to the list. The `Changed` event is raised by the `OnChanged` virtual method, which first checks whether the event is `null` (meaning that no handlers are present). The notion of raising an event is precisely equivalent to invoking the delegate represented by the event—thus, there are no special language constructs for raising events.

Clients react to events through **event handlers**. Event handlers are attached using the `+=` operator and removed using the `-=` operator. The following example attaches an event handler to the `Changed` event of a `List<string>`.

```
using System;

class Test
{
    static int changeCount;

    static void ListChanged(object sender, EventArgs e) {
        changeCount++;
    }

    static void Main() {
        List<string> names = new List<string>();
        names.Changed += new EventHandler(ListChanged);
        names.Add("Liz");
        names.Add("Martha");
        names.Add("Beth");
        Console.WriteLine(changeCount);           // Outputs "3"
    }
}
```

For advanced scenarios where control of the underlying storage of an event is desired, an event declaration can explicitly provide `add` and `remove` accessors, which are somewhat similar to the `set` accessor of a property.

Operators

An **operator** is a member that defines the meaning of applying a particular expression operator to instances of a class. Three kinds of operators can be defined: unary operators, binary operators, and conversion operators.

All operators must be declared as `public` and `static`.

The `List<T>` class declares two operators, `operator==` and `operator!=`, and thus gives new meaning to expressions that apply those operators to `List` instances. Specifically, the operators define equality of two `List<T>` instances as comparing each of the contained objects using their `Equals` methods. The following example uses the `==` operator to compare two `List<int>` instances.

```
using System;

class Test
{
    static void Main() {
        List<int> a = new List<int>();
        a.Add(1);
        a.Add(2);
        List<int> b = new List<int>();
        b.Add(1);
        b.Add(2);
        Console.WriteLine(a == b);      // Outputs "True"
        b.Add(3);
        Console.WriteLine(a == b);      // Outputs "False"
    }
}
```

The first `Console.WriteLine` outputs `True` because the two lists contain the same number of objects with the same values in the same order. Had `List<T>` not defined `operator==`, the first `Console.WriteLine` would have output `False` because `a` and `b` reference different `List<int>` instances.

Destructors

A **destructor** is a member that implements the actions required to destruct an instance of a class. Destructors cannot have parameters, they cannot have accessibility modifiers, and they cannot be invoked explicitly. The destructor for an instance is invoked automatically during garbage collection.

The garbage collector is allowed wide latitude in deciding when to collect objects and run destructors. Specifically, the timing of destructor invocations is not deterministic, and destructors may be executed on any thread. For these and other reasons, classes should implement destructors only when no other solutions are feasible.

The `using` statement provides a better approach to object destruction.

Structs

Like classes, **structs** are data structures that can contain data members and function members, but unlike classes, structs are value types and do not require heap allocation. A variable of a struct type directly stores the data of the struct, whereas a variable of a class type stores a reference to a dynamically allocated object. Struct types do not support user-specified inheritance, and all struct types implicitly inherit from type `object`.

Structs are particularly useful for small data structures that have value semantics. Complex numbers, points in a coordinate system, or key-value pairs in a dictionary are all good examples of structs. The use of structs rather than classes for small data structures can make a large difference in the number of memory allocations an application performs. For example, the following program creates and initializes an array of 100 points. With `Point` implemented as a class, 101 separate objects are instantiated—one for the array and one each for the 100 elements.

```

class Point
{
    public int x, y;

    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }
}

class Test
{
    static void Main() {
        Point[] points = new Point[100];
        for (int i = 0; i < 100; i++) points[i] = new Point(i, i);
    }
}

```

An alternative is to make `Point` a struct.

```

struct Point
{
    public int x, y;

    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }
}

```

Now, only one object is instantiated—the one for the array—and the `Point` instances are stored in-line in the array.

Struct constructors are invoked with the `new` operator, but that does not imply that memory is being allocated. Instead of dynamically allocating an object and returning a reference to it, a struct constructor simply returns the struct value itself (typically in a temporary location on the stack), and this value is then copied as necessary.

With classes, it is possible for two variables to reference the same object and thus possible for operations on one variable to affect the object referenced by the other variable. With structs, the variables each have their own copy of the data, and it is not possible for operations on one to affect the other. For example, the output produced by the following code fragment depends on whether `Point` is a class or a struct.

```

Point a = new Point(10, 10);
Point b = a;
a.x = 20;
Console.WriteLine(b.x);

```

If `Point` is a class, the output is `20` because `a` and `b` reference the same object. If `Point` is a struct, the output is `10` because the assignment of `a` to `b` creates a copy of the value, and this copy is unaffected by the subsequent assignment to `a.x`.

The previous example highlights two of the limitations of structs. First, copying an entire struct is typically less efficient than copying an object reference, so assignment and value parameter passing can be more expensive with structs than with reference types. Second, except for `ref` and `out` parameters, it is not possible to create references to structs, which rules out their usage in a number of situations.

Arrays

An **array** is a data structure that contains a number of variables that are accessed through computed indices. The variables contained in an array, also called the **elements** of the array, are all of the same type, and this type is called the **element type** of the array.

Array types are reference types, and the declaration of an array variable simply sets aside space for a reference to an array instance. Actual array instances are created dynamically at run-time using the `new` operator. The `new` operation specifies the **length** of the new array instance, which is then fixed for the lifetime of the instance. The indices of the elements of an array range from `0` to `Length - 1`. The `new` operator automatically initializes the elements of an array to their default value, which, for example, is zero for all numeric types and `null` for all reference types.

The following example creates an array of `int` elements, initializes the array, and prints out the contents of the array.

```
using System;

class Test
{
    static void Main() {
        int[] a = new int[10];
        for (int i = 0; i < a.Length; i++) {
            a[i] = i * i;
        }
        for (int i = 0; i < a.Length; i++) {
            Console.WriteLine("a[{0}] = {1}", i, a[i]);
        }
    }
}
```

This example creates and operates on a **single-dimensional array**. C# also supports **multi-dimensional arrays**. The number of dimensions of an array type, also known as the **rank** of the array type, is one plus the number of commas written between the square brackets of the array type. The following example allocates a one-dimensional, a two-dimensional, and a three-dimensional array.

```
int[] a1 = new int[10];
int[,] a2 = new int[10, 5];
int[,,] a3 = new int[10, 5, 2];
```

The `a1` array contains 10 elements, the `a2` array contains 50 (10×5) elements, and the `a3` array contains 100 ($10 \times 5 \times 2$) elements.

The element type of an array can be any type, including an array type. An array with elements of an array type is sometimes called a **jagged array** because the lengths of the element arrays do not all have to be the same. The following example allocates an array of arrays of `int`:

```
int[][] a = new int[3][];
a[0] = new int[10];
a[1] = new int[5];
a[2] = new int[20];
```

The first line creates an array with three elements, each of type `int[]` and each with an initial value of `null`. The subsequent lines then initialize the three elements with references to individual array instances of varying lengths.

The `new` operator permits the initial values of the array elements to be specified using an **array initializer**, which is a list of expressions written between the delimiters `{` and `}`. The following example allocates and

initializes an `int[]` with three elements.

```
int[] a = new int[] {1, 2, 3};
```

Note that the length of the array is inferred from the number of expressions between `{` and `}`. Local variable and field declarations can be shortened further such that the array type does not have to be restated.

```
int[] a = {1, 2, 3};
```

Both of the previous examples are equivalent to the following:

```
int[] t = new int[3];  
t[0] = 1;  
t[1] = 2;  
t[2] = 3;  
int[] a = t;
```

Interfaces

An *interface* defines a contract that can be implemented by classes and structs. An interface can contain methods, properties, events, and indexers. An interface does not provide implementations of the members it defines—it merely specifies the members that must be supplied by classes or structs that implement the interface.

Interfaces may employ *multiple inheritance*. In the following example, the interface `IComboBox` inherits from both `ITextBox` and `IListBox`.

```
interface IControl  
{  
    void Paint();  
}  
  
interface ITextBox: IControl  
{  
    void SetText(string text);  
}  
  
interface IListBox: IControl  
{  
    void SetItems(string[] items);  
}  
  
interface IComboBox: ITextBox, IListBox {}
```

Classes and structs can implement multiple interfaces. In the following example, the class `EditBox` implements both `IControl` and `IDataBound`.

```
interface IDataBound
{
    void Bind(Binder b);
}

public class EditBox: IControl, IDataBound
{
    public void Paint() {...}
    public void Bind(Binder b) {...}
}
```

When a class or struct implements a particular interface, instances of that class or struct can be implicitly converted to that interface type. For example

```
EditBox editBox = new EditBox();
IControl control = editBox;
IDataBound dataBound = editBox;
```

In cases where an instance is not statically known to implement a particular interface, dynamic type casts can be used. For example, the following statements use dynamic type casts to obtain an object's `IControl` and `IDataBound` interface implementations. Because the actual type of the object is `EditBox`, the casts succeed.

```
object obj = new EditBox();
IControl control = (IControl)obj;
IDataBound dataBound = (IDataBound)obj;
```

In the previous `EditBox` class, the `Paint` method from the `IControl` interface and the `Bind` method from the `IDataBound` interface are implemented using `public` members. C# also supports *explicit interface member implementations*, using which the class or struct can avoid making the members `public`. An explicit interface member implementation is written using the fully qualified interface member name. For example, the `EditBox` class could implement the `IControl.Paint` and `IDataBound.Bind` methods using explicit interface member implementations as follows.

```
public class EditBox: IControl, IDataBound
{
    void IControl.Paint() {...}
    void IDataBound.Bind(Binder b) {...}
}
```

Explicit interface members can only be accessed via the interface type. For example, the implementation of `IControl.Paint` provided by the previous `EditBox` class can only be invoked by first converting the `EditBox` reference to the `IControl` interface type.

```
EditBox editBox = new EditBox();
editBox.Paint();           // Error, no such method
IControl control = editBox;
control.Paint();           // Ok
```

Enums

An *enum type* is a distinct value type with a set of named constants. The following example declares and uses an enum type named `Color` with three constant values, `Red`, `Green`, and `Blue`.

```

using System;

enum Color
{
    Red,
    Green,
    Blue
}

class Test
{
    static void PrintColor(Color color) {
        switch (color) {
            case Color.Red:
                Console.WriteLine("Red");
                break;
            case Color.Green:
                Console.WriteLine("Green");
                break;
            case Color.Blue:
                Console.WriteLine("Blue");
                break;
            default:
                Console.WriteLine("Unknown color");
                break;
        }
    }

    static void Main() {
        Color c = Color.Red;
        PrintColor(c);
        PrintColor(Color.Blue);
    }
}

```

Each enum type has a corresponding integral type called the *underlying type* of the enum type. An enum type that does not explicitly declare an underlying type has an underlying type of `int`. An enum type's storage format and range of possible values are determined by its underlying type. The set of values that an enum type can take on is not limited by its enum members. In particular, any value of the underlying type of an enum can be cast to the enum type and is a distinct valid value of that enum type.

The following example declares an enum type named `Alignment` with an underlying type of `sbyte`.

```

enum Alignment: sbyte
{
    Left = -1,
    Center = 0,
    Right = 1
}

```

As shown by the previous example, an enum member declaration can include a constant expression that specifies the value of the member. The constant value for each enum member must be in the range of the underlying type of the enum. When an enum member declaration does not explicitly specify a value, the member is given the value zero (if it is the first member in the enum type) or the value of the textually preceding enum member plus one.

Enum values can be converted to integral values and vice versa using type casts. For example

```

int i = (int)Color.Blue;      // int i = 2;
Color c = (Color)2;          // Color c = Color.Blue;

```

The default value of any enum type is the integral value zero converted to the enum type. In cases where variables are automatically initialized to a default value, this is the value given to variables of enum types. In order for the default value of an enum type to be easily available, the literal `0` implicitly converts to any enum type. Thus, the following is permitted.

```
Color c = 0;
```

Delegates

A *delegate type* represents references to methods with a particular parameter list and return type. Delegates make it possible to treat methods as entities that can be assigned to variables and passed as parameters. Delegates are similar to the concept of function pointers found in some other languages, but unlike function pointers, delegates are object-oriented and type-safe.

The following example declares and uses a delegate type named `Function`.

```
using System;

delegate double Function(double x);

class Multiplier
{
    double factor;

    public Multiplier(double factor) {
        this.factor = factor;
    }

    public double Multiply(double x) {
        return x * factor;
    }
}

class Test
{
    static double Square(double x) {
        return x * x;
    }

    static double[] Apply(double[] a, Function f) {
        double[] result = new double[a.Length];
        for (int i = 0; i < a.Length; i++) result[i] = f(a[i]);
        return result;
    }

    static void Main() {
        double[] a = {0.0, 0.5, 1.0};
        double[] squares = Apply(a, Square);
        double[] sines = Apply(a, Math.Sin);
        Multiplier m = new Multiplier(2.0);
        double[] doubles = Apply(a, m.Multiply);
    }
}
```

An instance of the `Function` delegate type can reference any method that takes a `double` argument and returns a `double` value. The `Apply` method applies a given `Function` to the elements of a `double[]`, returning a `double[]` with the results. In the `Main` method, `Apply` is used to apply three different functions to a `double[]`.

A delegate can reference either a static method (such as `Square` or `Math.Sin` in the previous example) or an instance method (such as `m.Multiply` in the previous example). A delegate that references an instance method

also references a particular object, and when the instance method is invoked through the delegate, that object becomes `this` in the invocation.

Delegates can also be created using anonymous functions, which are "inline methods" that are created on the fly. Anonymous functions can see the local variables of the surrounding methods. Thus, the multiplier example above can be written more easily without using a `Multiplier` class:

```
double[] doubles = Apply(a, (double x) => x * 2.0);
```

An interesting and useful property of a delegate is that it does not know or care about the class of the method it references; all that matters is that the referenced method has the same parameters and return type as the delegate.

Attributes

Types, members, and other entities in a C# program support modifiers that control certain aspects of their behavior. For example, the accessibility of a method is controlled using the `public`, `protected`, `internal`, and `private` modifiers. C# generalizes this capability such that user-defined types of declarative information can be attached to program entities and retrieved at run-time. Programs specify this additional declarative information by defining and using *attributes*.

The following example declares a `HelpAttribute` attribute that can be placed on program entities to provide links to their associated documentation.

```
using System;

public class HelpAttribute: Attribute
{
    string url;
    string topic;

    public HelpAttribute(string url) {
        this.url = url;
    }

    public string Url {
        get { return url; }
    }

    public string Topic {
        get { return topic; }
        set { topic = value; }
    }
}
```

All attribute classes derive from the `System.Attribute` base class provided by the .NET Framework. Attributes can be applied by giving their name, along with any arguments, inside square brackets just before the associated declaration. If an attribute's name ends in `Attribute`, that part of the name can be omitted when the attribute is referenced. For example, the `HelpAttribute` attribute can be used as follows.

```
[Help("http://msdn.microsoft.com/.../MyClass.htm")]
public class Widget
{
    [Help("http://msdn.microsoft.com/.../MyClass.htm", Topic = "Display")]
    public void Display(string text) {}
}
```


This example attaches a `HelpAttribute` to the `Widget` class and another `HelpAttribute` to the `Display` method in the class. The public constructors of an attribute class control the information that must be provided when the attribute is attached to a program entity. Additional information can be provided by referencing public read-write properties of the attribute class (such as the reference to the `Topic` property previously).

The following example shows how attribute information for a given program entity can be retrieved at run-time using reflection.

```
using System;
using System.Reflection;

class Test
{
    static void ShowHelp(MemberInfo member) {
        HelpAttribute a = Attribute.GetCustomAttribute(member,
            typeof(HelpAttribute)) as HelpAttribute;
        if (a == null) {
            Console.WriteLine("No help for {0}", member);
        }
        else {
            Console.WriteLine("Help for {0}:", member);
            Console.WriteLine("  Url={0}, Topic={1}", a.Url, a.Topic);
        }
    }

    static void Main() {
        ShowHelp(typeof(Widget));
        ShowHelp(typeof(Widget).GetMethod("Display"));
    }
}
```

When a particular attribute is requested through reflection, the constructor for the attribute class is invoked with the information provided in the program source, and the resulting attribute instance is returned. If additional information was provided through properties, those properties are set to the given values before the attribute instance is returned.

Lexical structure

12/28/2021 • 33 minutes to read • [Edit Online](#)

Programs

A C# **program** consists of one or more **source files**, known formally as **compilation units** ([Compilation units](#)). A source file is an ordered sequence of Unicode characters. Source files typically have a one-to-one correspondence with files in a file system, but this correspondence is not required. For maximal portability, it is recommended that files in a file system be encoded with the UTF-8 encoding.

Conceptually speaking, a program is compiled using three steps:

1. Transformation, which converts a file from a particular character repertoire and encoding scheme into a sequence of Unicode characters.
2. Lexical analysis, which translates a stream of Unicode input characters into a stream of tokens.
3. Syntactic analysis, which translates the stream of tokens into executable code.

Grammars

This specification presents the syntax of the C# programming language using two grammars. The **lexical grammar** ([Lexical grammar](#)) defines how Unicode characters are combined to form line terminators, white space, comments, tokens, and pre-processing directives. The **syntactic grammar** ([Syntactic grammar](#)) defines how the tokens resulting from the lexical grammar are combined to form C# programs.

Grammar notation

The lexical and syntactic grammars are presented in Backus-Naur form using the notation of the ANTLR grammar tool.

Lexical grammar

The lexical grammar of C# is presented in [Lexical analysis](#), [Tokens](#), and [Pre-processing directives](#). The terminal symbols of the lexical grammar are the characters of the Unicode character set, and the lexical grammar specifies how characters are combined to form tokens ([Tokens](#)), white space ([White space](#)), comments ([Comments](#)), and pre-processing directives ([Pre-processing directives](#)).

Every source file in a C# program must conform to the *input* production of the lexical grammar ([Lexical analysis](#)).

Syntactic grammar

The syntactic grammar of C# is presented in the chapters and appendices that follow this chapter. The terminal symbols of the syntactic grammar are the tokens defined by the lexical grammar, and the syntactic grammar specifies how tokens are combined to form C# programs.

Every source file in a C# program must conform to the *compilation_unit* production of the syntactic grammar ([Compilation units](#)).

Lexical analysis

The *input* production defines the lexical structure of a C# source file. Each source file in a C# program must conform to this lexical grammar production.

```

input
    : input_section?
    ;

input_section
    : input_section_part+
    ;

input_section_part
    : input_element* new_line
    | pp_directive
    ;

input_element
    : whitespace
    | comment
    | token
    ;

```

Five basic elements make up the lexical structure of a C# source file: Line terminators ([Line terminators](#)), white space ([White space](#)), comments ([Comments](#)), tokens ([Tokens](#)), and pre-processing directives ([Pre-processing directives](#)). Of these basic elements, only tokens are significant in the syntactic grammar of a C# program ([Syntactic grammar](#)).

The lexical processing of a C# source file consists of reducing the file into a sequence of tokens which becomes the input to the syntactic analysis. Line terminators, white space, and comments can serve to separate tokens, and pre-processing directives can cause sections of the source file to be skipped, but otherwise these lexical elements have no impact on the syntactic structure of a C# program.

In the case of interpolated string literals ([Interpolated string literals](#)) a single token is initially produced by lexical analysis, but is broken up into several input elements which are repeatedly subjected to lexical analysis until all interpolated string literals have been resolved. The resulting tokens then serve as input to the syntactic analysis.

When several lexical grammar productions match a sequence of characters in a source file, the lexical processing always forms the longest possible lexical element. For example, the character sequence `//` is processed as the beginning of a single-line comment because that lexical element is longer than a single `/` token.

Line terminators

Line terminators divide the characters of a C# source file into lines.

```

new_line
    : '<Carriage return character (U+000D)>'
    | '<Line feed character (U+000A)>'
    | '<Carriage return character (U+000D) followed by line feed character (U+000A)>'
    | '<Next line character (U+0085)>'
    | '<Line separator character (U+2028)>'
    | '<Paragraph separator character (U+2029)>'
    ;

```

For compatibility with source code editing tools that add end-of-file markers, and to enable a source file to be viewed as a sequence of properly terminated lines, the following transformations are applied, in order, to every source file in a C# program:

- If the last character of the source file is a Control-Z character (`U+001A`), this character is deleted.
- A carriage-return character (`U+000D`) is added to the end of the source file if that source file is non-empty and if the last character of the source file is not a carriage return (`U+000D`), a line feed (`U+000A`), a line separator (`U+2028`), or a paragraph separator (`U+2029`).

Comments

Two forms of comments are supported: single-line comments and delimited comments. *Single-line comments* start with the characters `//` and extend to the end of the source line. *Delimited comments* start with the characters `/*` and end with the characters `*/`. Delimited comments may span multiple lines.

```
comment
  : single_line_comment
  | delimited_comment
  ;

single_line_comment
  : '//' input_character*
  ;

input_character
  : '<Any Unicode character except a new_line_character>'
  ;

new_line_character
  : '<Carriage return character (U+000D)>'
  | '<Line feed character (U+000A)>'
  | '<Next line character (U+0085)>'
  | '<Line separator character (U+2028)>'
  | '<Paragraph separator character (U+2029)>'
  ;

delimited_comment
  : '/*' delimited_comment_section* asterisk+ '/'
  ;

delimited_comment_section
  : '/'
  | asterisk* not_slash_or_asterisk
  ;

asterisk
  : '*'
  ;

not_slash_or_asterisk
  : '<Any Unicode character except / or *>'
  ;
```

Comments do not nest. The character sequences `/*` and `*/` have no special meaning within a `//` comment, and the character sequences `//` and `/*` have no special meaning within a delimited comment.

Comments are not processed within character and string literals.

The example

```
/* Hello, world program
   This program writes "hello, world" to the console
*/
class Hello
{
    static void Main() {
        System.Console.WriteLine("hello, world");
    }
}
```

includes a delimited comment.

The example

```
// Hello, world program
// This program writes "hello, world" to the console
//
class Hello // any name will do for this class
{
    static void Main() { // this method must be named "Main"
        System.Console.WriteLine("hello, world");
    }
}
```

shows several single-line comments.

White space

White space is defined as any character with Unicode class Zs (which includes the space character) as well as the horizontal tab character, the vertical tab character, and the form feed character.

```
whitespace
: '<Any character with Unicode class Zs>'
| '<Horizontal tab character (U+0009)>'
| '<Vertical tab character (U+000B)>'
| '<Form feed character (U+000C)>'
;
```

Tokens

There are several kinds of tokens: identifiers, keywords, literals, operators, and punctuators. White space and comments are not tokens, though they act as separators for tokens.

```
token
: identifier
| keyword
| integer_literal
| real_literal
| character_literal
| string_literal
| interpolated_string_literal
| operator_or_punctuator
;
```

Unicode character escape sequences

A Unicode character escape sequence represents a Unicode character. Unicode character escape sequences are processed in identifiers ([Identifiers](#)), character literals ([Character literals](#)), and regular string literals ([String literals](#)). A Unicode character escape is not processed in any other location (for example, to form an operator, punctuator, or keyword).

```
unicode_escape_sequence
: '\\u' hex_digit hex_digit hex_digit hex_digit
| '\\U' hex_digit hex_digit hex_digit hex_digit hex_digit hex_digit hex_digit hex_digit
;
```

A Unicode escape sequence represents the single Unicode character formed by the hexadecimal number following the "`\u`" or "`\U`" characters. Since C# uses a 16-bit encoding of Unicode code points in characters and string values, a Unicode character in the range U+10000 to U+10FFFF is not permitted in a character literal and is represented using a Unicode surrogate pair in a string literal. Unicode characters with code points above 0x10FFFF are not supported.

Multiple translations are not performed. For instance, the string literal "\u005Cu005C" is equivalent to "\u005C" rather than "\\". The Unicode value \u005C is the character "\\".

The example

```
class Class1
{
    static void Test(bool \u0066) {
        char c = '\u0066';
        if (\u0066)
            System.Console.WriteLine(c.ToString());
    }
}
```

shows several uses of \u0066, which is the escape sequence for the letter "f". The program is equivalent to

```
class Class1
{
    static void Test(bool f) {
        char c = 'f';
        if (f)
            System.Console.WriteLine(c.ToString());
    }
}
```

Identifiers

The rules for identifiers given in this section correspond exactly to those recommended by the Unicode Standard Annex 31, except that underscore is allowed as an initial character (as is traditional in the C programming language), Unicode escape sequences are permitted in identifiers, and the "@" character is allowed as a prefix to enable keywords to be used as identifiers.

```

identifier
    : available_identifier
    | '@' identifier_or_keyword
    ;

available_identifier
    : '<An identifier_or_keyword that is not a keyword>'
    ;

identifier_or_keyword
    : identifier_start_character identifier_part_character*
    ;

identifier_start_character
    : letter_character
    | '_'
    ;

identifier_part_character
    : letter_character
    | decimal_digit_character
    | connecting_character
    | combining_character
    | formatting_character
    ;

letter_character
    : '<A Unicode character of classes Lu, Ll, Lt, Lm, Lo, or Nl>'
    | '<A unicode_escape_sequence representing a character of classes Lu, Ll, Lt, Lm, Lo, or Nl>'
    ;

combining_character
    : '<A Unicode character of classes Mn or Mc>'
    | '<A unicode_escape_sequence representing a character of classes Mn or Mc>'
    ;

decimal_digit_character
    : '<A Unicode character of the class Nd>'
    | '<A unicode_escape_sequence representing a character of the class Nd>'
    ;

connecting_character
    : '<A Unicode character of the class Pc>'
    | '<A unicode_escape_sequence representing a character of the class Pc>'
    ;

formatting_character
    : '<A Unicode character of the class Cf>'
    | '<A unicode_escape_sequence representing a character of the class Cf>'
    ;

```

For information on the Unicode character classes mentioned above, see The Unicode Standard, Version 3.0, section 4.5.

Examples of valid identifiers include "identifier1", "_identifier2", and "@if".

An identifier in a conforming program must be in the canonical format defined by Unicode Normalization Form C, as defined by Unicode Standard Annex 15. The behavior when encountering an identifier not in Normalization Form C is implementation-defined; however, a diagnostic is not required.

The prefix "@" enables the use of keywords as identifiers, which is useful when interfacing with other programming languages. The character @ is not actually part of the identifier, so the identifier might be seen in other languages as a normal identifier, without the prefix. An identifier with an @ prefix is called a *verbatim identifier*. Use of the @ prefix for identifiers that are not keywords is permitted, but strongly discouraged as a

matter of style.

The example:

```
class @class
{
    public static void @static(bool @bool) {
        if (@bool)
            System.Console.WriteLine("true");
        else
            System.Console.WriteLine("false");
    }
}

class Class1
{
    static void M() {
        cl\u0061ss.st\u0061tic(true);
    }
}
```

defines a class named "`class`" with a static method named "`static`" that takes a parameter named "`bool`". Note that since Unicode escapes are not permitted in keywords, the token "`cl\u0061ss`" is an identifier, and is the same identifier as "`@class`".

Two identifiers are considered the same if they are identical after the following transformations are applied, in order:

- The prefix "`@`", if used, is removed.
- Each *unicode_escape_sequence* is transformed into its corresponding Unicode character.
- Any *formatting_characters* are removed.

Identifiers containing two consecutive underscore characters (`U+005F`) are reserved for use by the implementation. For example, an implementation might provide extended keywords that begin with two underscores.

Keywords

A **keyword** is an identifier-like sequence of characters that is reserved, and cannot be used as an identifier except when prefaced by the `@` character.

```
keyword
: 'abstract' | 'as'      | 'base'      | 'bool'      | 'break'
| 'byte'      | 'case'      | 'catch'     | 'char'      | 'checked'
| 'class'     | 'const'     | 'continue'  | 'decimal'   | 'default'
| 'delegate'  | 'do'        | 'double'    | 'else'      | 'enum'
| 'event'     | 'explicit'  | 'extern'    | 'false'     | 'finally'
| 'fixed'     | 'float'     | 'for'       | 'foreach'   | 'goto'
| 'if'        | 'implicit'  | 'in'        | 'int'       | 'interface'
| 'internal'  | 'is'        | 'lock'      | 'long'      | 'namespace'
| 'new'       | 'null'      | 'object'    | 'operator'  | 'out'
| 'override'  | 'params'    | 'private'   | 'protected' | 'public'
| 'readonly'  | 'ref'       | 'return'    | 'sbyte'     | 'sealed'
| 'short'     | 'sizeof'    | 'stackalloc'| 'static'    | 'string'
| 'struct'    | 'switch'    | 'this'      | 'throw'     | 'true'
| 'try'       | 'typeof'    | 'uint'      | 'ulong'     | 'unchecked'
| 'unsafe'    | 'ushort'    | 'using'     | 'virtual'   | 'void'
| 'volatile'  | 'while'
;
```

In some places in the grammar, specific identifiers have special meaning, but are not keywords. Such identifiers

are sometimes referred to as "contextual keywords". For example, within a property declaration, the "`get`" and "`set`" identifiers have special meaning ([Accessors](#)). An identifier other than `get` or `set` is never permitted in these locations, so this use does not conflict with a use of these words as identifiers. In other cases, such as with the identifier "`var`" in implicitly typed local variable declarations ([Local variable declarations](#)), a contextual keyword can conflict with declared names. In such cases, the declared name takes precedence over the use of the identifier as a contextual keyword.

Literals

A *literal* is a source code representation of a value.

```
literal
: boolean_literal
| integer_literal
| real_literal
| character_literal
| string_literal
| null_literal
;
```

Boolean literals

There are two boolean literal values: `true` and `false`.

```
boolean_literal
: 'true'
| 'false'
;
```

The type of a *boolean_literal* is `bool`.

Integer literals

Integer literals are used to write values of types `int`, `uint`, `long`, and `ulong`. Integer literals have two possible forms: decimal and hexadecimal.

```
integer_literal
: decimal_integer_literal
| hexadecimal_integer_literal
;

decimal_integer_literal
: decimal_digit+ integer_type_suffix?
;

decimal_digit
: '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'
;

integer_type_suffix
: 'U' | 'u' | 'L' | 'l' | 'UL' | 'Ul' | 'uL' | 'ul' | 'LU' | 'Lu' | 'LU' | 'lu'
;

hexadecimal_integer_literal
: '0x' hex_digit+ integer_type_suffix?
| '0X' hex_digit+ integer_type_suffix?
;

hex_digit
: '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'
| 'A' | 'B' | 'C' | 'D' | 'E' | 'F' | 'a' | 'b' | 'c' | 'd' | 'e' | 'f';
```

The type of an integer literal is determined as follows:

- If the literal has no suffix, it has the first of these types in which its value can be represented: `int`, `uint`, `long`, `ulong`.
- If the literal is suffixed by `U` or `u`, it has the first of these types in which its value can be represented: `uint`, `ulong`.
- If the literal is suffixed by `L` or `l`, it has the first of these types in which its value can be represented: `long`, `ulong`.
- If the literal is suffixed by `UL`, `Ul`, `uL`, `ul`, `LU`, `Lu`, `lU`, or `lu`, it is of type `ulong`.

If the value represented by an integer literal is outside the range of the `ulong` type, a compile-time error occurs.

As a matter of style, it is suggested that "`L`" be used instead of "`l`" when writing literals of type `long`, since it is easy to confuse the letter "`l`" with the digit "`1`".

To permit the smallest possible `int` and `long` values to be written as decimal integer literals, the following two rules exist:

- When a *decimal_integer_literal* with the value 2147483648 (2^{31}) and no *integer_type_suffix* appears as the token immediately following a unary minus operator token ([Unary minus operator](#)), the result is a constant of type `int` with the value -2147483648 (-2^{31}). In all other situations, such a *decimal_integer_literal* is of type `uint`.
- When a *decimal_integer_literal* with the value 9223372036854775808 (2^{63}) and no *integer_type_suffix* or the *integer_type_suffix* `L` or `l` appears as the token immediately following a unary minus operator token ([Unary minus operator](#)), the result is a constant of type `long` with the value -9223372036854775808 (-2^{63}). In all other situations, such a *decimal_integer_literal* is of type `ulong`.

Real literals

Real literals are used to write values of types `float`, `double`, and `decimal`.

```
real_literal
: decimal_digit+ '.' decimal_digit+ exponent_part? real_type_suffix?
| '.' decimal_digit+ exponent_part? real_type_suffix?
| decimal_digit+ exponent_part real_type_suffix?
| decimal_digit+ real_type_suffix
;

exponent_part
: 'e' sign? decimal_digit+
| 'E' sign? decimal_digit+
;

sign
: '+'
| '-'
;

real_type_suffix
: 'F' | 'f' | 'D' | 'd' | 'M' | 'm'
;
```

If no *real_type_suffix* is specified, the type of the real literal is `double`. Otherwise, the real type suffix determines the type of the real literal, as follows:

- A real literal suffixed by `F` or `f` is of type `float`. For example, the literals `1f`, `1.5f`, `1e10f`, and `123.456F` are all of type `float`.
- A real literal suffixed by `D` or `d` is of type `double`. For example, the literals `1d`, `1.5d`, `1e10d`, and `123.456D` are all of type `double`.
- A real literal suffixed by `M` or `m` is of type `decimal`. For example, the literals `1m`, `1.5m`, `1e10m`, and

`123.456M` are all of type `decimal`. This literal is converted to a `decimal` value by taking the exact value, and, if necessary, rounding to the nearest representable value using banker's rounding ([The decimal type](#)). Any scale apparent in the literal is preserved unless the value is rounded or the value is zero (in which latter case the sign and scale will be 0). Hence, the literal `2.900m` will be parsed to form the decimal with sign `0`, coefficient `2900`, and scale `3`.

If the specified literal cannot be represented in the indicated type, a compile-time error occurs.

The value of a real literal of type `float` or `double` is determined by using the IEEE "round to nearest" mode.

Note that in a real literal, decimal digits are always required after the decimal point. For example, `1.3F` is a real literal but `1.F` is not.

Character literals

A character literal represents a single character, and usually consists of a character in quotes, as in `'a'`.

Note: The ANTLR grammar notation makes the following confusing! In ANTLR, when you write `\'` it stands for a single quote `'`. And when you write `\\` it stands for a single backslash `\`. Therefore the first rule for a character literal means it starts with a single quote, then a character, then a single quote. And the eleven possible escape sequences are `\'`, `\"`, `\\`, `\0`, `\a`, `\b`, `\f`, `\n`, `\r`, `\t`, `\v`.

```
character_literal
: '\'' character '\''
;

character
: single_character
| simple_escape_sequence
| hexadecimal_escape_sequence
| unicode_escape_sequence
;

single_character
: '<Any character except \' (U+0027), \'\' (U+005C), and new_line_character>'
;

simple_escape_sequence
: '\\\'' | '\\\"' | '\\\\' | '\\0' | '\\a' | '\\b' | '\\f' | '\\n' | '\\r' | '\\t' | '\\v'
;

hexadecimal_escape_sequence
: '\\x' hex_digit hex_digit? hex_digit? hex_digit?;
```

A character that follows a backslash character (`\`) in a *character* must be one of the following characters: `'`, `"`, `\`, `0`, `a`, `b`, `f`, `n`, `r`, `t`, `u`, `U`, `x`, `v`. Otherwise, a compile-time error occurs.

A hexadecimal escape sequence represents a single Unicode character, with the value formed by the hexadecimal number following `\x`.

If the value represented by a character literal is greater than `U+FFFF`, a compile-time error occurs.

A Unicode character escape sequence ([Unicode character escape sequences](#)) in a character literal must be in the range `U+0000` to `U+FFFF`.

A simple escape sequence represents a Unicode character encoding, as described in the table below.

ESCAPE SEQUENCE	CHARACTER NAME	UNICODE ENCODING
<code>\'</code>	Single quote	<code>0x0027</code>

ESCAPE SEQUENCE	CHARACTER NAME	UNICODE ENCODING
<code>\"</code>	Double quote	<code>0x0022</code>
<code>\\</code>	Backslash	<code>0x005C</code>
<code>\0</code>	Null	<code>0x0000</code>
<code>\a</code>	Alert	<code>0x0007</code>
<code>\b</code>	Backspace	<code>0x0008</code>
<code>\f</code>	Form feed	<code>0x000C</code>
<code>\n</code>	New line	<code>0x000A</code>
<code>\r</code>	Carriage return	<code>0x000D</code>
<code>\t</code>	Horizontal tab	<code>0x0009</code>
<code>\v</code>	Vertical tab	<code>0x000B</code>

The type of a *character_literal* is `char`.

String literals

C# supports two forms of string literals: *regular string literals* and *verbatim string literals*.

A regular string literal consists of zero or more characters enclosed in double quotes, as in `"hello"`, and may include both simple escape sequences (such as `\t` for the tab character), and hexadecimal and Unicode escape sequences.

A verbatim string literal consists of an `@` character followed by a double-quote character, zero or more characters, and a closing double-quote character. A simple example is `@"hello"`. In a verbatim string literal, the characters between the delimiters are interpreted verbatim, the only exception being a *quote_escape_sequence*. In particular, simple escape sequences, and hexadecimal and Unicode escape sequences are not processed in verbatim string literals. A verbatim string literal may span multiple lines.

```

string_literal
: regular_string_literal
| verbatim_string_literal
;

regular_string_literal
: '"' regular_string_literal_character* '"'
;

regular_string_literal_character
: single_regular_string_literal_character
| simple_escape_sequence
| hexadecimal_escape_sequence
| unicode_escape_sequence
;

single_regular_string_literal_character
: '<Any character except " (U+0022), \\ (U+005C), and new_line_character>'
;

verbatim_string_literal
: '@"' verbatim_string_literal_character* '"'
;

verbatim_string_literal_character
: single_verbatim_string_literal_character
| quote_escape_sequence
;

single_verbatim_string_literal_character
: '<any character except ">'
;

quote_escape_sequence
: '""'
;

```

A character that follows a backslash character (`\`) in a *regular_string_literal_character* must be one of the following characters: `'`, `"`, `\`, `0`, `a`, `b`, `f`, `n`, `r`, `t`, `u`, `U`, `x`, `v`. Otherwise, a compile-time error occurs.

The example

```

string a = "hello, world";           // hello, world
string b = @"hello, world";          // hello, world

string c = "hello \t world";          // hello      world
string d = @"hello \t world";          // hello \t world

string e = "Joe said \"Hello\" to me"; // Joe said "Hello" to me
string f = @"Joe said ""Hello"" to me"; // Joe said "Hello" to me

string g = "\\server\\share\\file.txt"; // \\server\share\file.txt
string h = @"\\server\share\file.txt";  // \\server\share\file.txt

string i = "one\r\ntwo\r\nthree";
string j = @"one
two
three";

```

shows a variety of string literals. The last string literal, `j`, is a verbatim string literal that spans multiple lines. The characters between the quotation marks, including white space such as new line characters, are preserved verbatim.

Since a hexadecimal escape sequence can have a variable number of hex digits, the string literal `"\x123"` contains a single character with hex value 123. To create a string containing the character with hex value 12 followed by the character 3, one could write `"\x00123"` or `"\x12" + "3"` instead.

The type of a *string literal* is `string`.

Each string literal does not necessarily result in a new string instance. When two or more string literals that are equivalent according to the string equality operator ([String equality operators](#)) appear in the same program, these string literals refer to the same string instance. For instance, the output produced by

```
class Test
{
    static void Main() {
        object a = "hello";
        object b = "hello";
        System.Console.WriteLine(a == b);
    }
}
```

is `True` because the two literals refer to the same string instance.

Interpolated string literals

Interpolated string literals are similar to string literals, but contain holes delimited by `{` and `}`, wherein expressions can occur. At runtime, the expressions are evaluated with the purpose of having their textual forms substituted into the string at the place where the hole occurs. The syntax and semantics of string interpolation are described in section ([Interpolated strings](#)).

Like string literals, interpolated string literals can be either regular or verbatim. Interpolated regular string literals are delimited by `$"` and `"`, and interpolated verbatim string literals are delimited by `$@"` and `"`.

Like other literals, lexical analysis of an interpolated string literal initially results in a single token, as per the grammar below. However, before syntactic analysis, the single token of an interpolated string literal is broken into several tokens for the parts of the string enclosing the holes, and the input elements occurring in the holes are lexically analysed again. This may in turn produce more interpolated string literals to be processed, but, if lexically correct, will eventually lead to a sequence of tokens for syntactic analysis to process.

```
interpolated_string_literal
: '$' interpolated_regular_string_literal
| '$' interpolated_verbatim_string_literal
;

interpolated_regular_string_literal
: interpolated_regular_string_whole
| interpolated_regular_string_start interpolated_regular_string_literal_body
interpolated_regular_string_end
;

interpolated_regular_string_literal_body
: regular_balanced_text
| interpolated_regular_string_literal_body interpolated_regular_string_mid regular_balanced_text
;

interpolated_regular_string_whole
: ''' interpolated_regular_string_character* '''
;

interpolated_regular_string_start
: ''' interpolated_regular_string_character* '{'
;

interpolated_regular_string_mid
: interpolated_regular_string_character* '}'
;
```

```

: interpolation_format? '{' interpolated_regular_string_characters_after_brace? '{'
;

interpolated_regular_string_end
: interpolation_format? '{' interpolated_regular_string_characters_after_brace? '{'
;

interpolated_regular_string_characters_after_brace
: interpolated_regular_string_character_no_brace
| interpolated_regular_string_characters_after_brace interpolated_regular_string_character
;

interpolated_regular_string_character
: single_interpolated_regular_string_character
| simple_escape_sequence
| hexadecimal_escape_sequence
| unicode_escape_sequence
| open_brace_escape_sequence
| close_brace_escape_sequence
;

interpolated_regular_string_character_no_brace
: '<Any interpolated_regular_string_character except close_brace_escape_sequence and any
hexadecimal_escape_sequence or unicode_escape_sequence designating } (U+007D)>'
;

single_interpolated_regular_string_character
: '<Any character except \" (U+0022), \\ (U+005C), { (U+007B), } (U+007D), and new_line_character>'
;

open_brace_escape_sequence
: '{{'
;

close_brace_escape_sequence
: '}}'
;

regular_balanced_text
: regular_balanced_text_part+
;

regular_balanced_text_part
: single_regular_balanced_text_character
| delimited_comment
| '@' identifier_or_keyword
| string_literal
| interpolated_string_literal
| '(' regular_balanced_text ')'
| '[' regular_balanced_text ']'
| '{' regular_balanced_text '}'
;

single_regular_balanced_text_character
: '<Any character except / (U+002F), @ (U+0040), \" (U+0022), $ (U+0024), ( (U+0028), ) (U+0029), [
(U+005B), ] (U+005D), { (U+007B), } (U+007D) and new_line_character>'
| '</ (U+002F), if not directly followed by / (U+002F) or * (U+002A)>'
;

interpolation_format
: ':' interpolation_format_character+
;

interpolation_format_character
: '<Any character except \" (U+0022), : (U+003A), { (U+007B) and } (U+007D)>'
;

interpolated_verbatim_string_literal
: interpolated_verbatim_string_whole

```

```

    | interpolated_verbatim_string_start interpolated_verbatim_string_literal_body
interpolated_verbatim_string_end
;

interpolated_verbatim_string_literal_body
: verbatim_balanced_text
| interpolated_verbatim_string_literal_body interpolated_verbatim_string_mid verbatim_balanced_text
;

interpolated_verbatim_string_whole
: '@"' interpolated_verbatim_string_character* '"'
;

interpolated_verbatim_string_start
: '@"' interpolated_verbatim_string_character* '{'
;

interpolated_verbatim_string_mid
: interpolation_format? '}' interpolated_verbatim_string_characters_after_brace? '{'
;

interpolated_verbatim_string_end
: interpolation_format? '}' interpolated_verbatim_string_characters_after_brace? '"'
;

interpolated_verbatim_string_characters_after_brace
: interpolated_verbatim_string_character_no_brace
| interpolated_verbatim_string_characters_after_brace interpolated_verbatim_string_character
;

interpolated_verbatim_string_character
: single_interpolated_verbatim_string_character
| quote_escape_sequence
| open_brace_escape_sequence
| close_brace_escape_sequence
;

interpolated_verbatim_string_character_no_brace
: '<Any interpolated_verbatim_string_character except close_brace_escape_sequence>'
;

single_interpolated_verbatim_string_character
: '<Any character except \" (U+0022), { (U+007B) and } (U+007D)>'
;

verbatim_balanced_text
: verbatim_balanced_text_part+
;

verbatim_balanced_text_part
: single_verbatim_balanced_text_character
| comment
| '@' identifier_or_keyword
| string_literal
| interpolated_string_literal
| '(' verbatim_balanced_text ')'
| '[' verbatim_balanced_text ']'
| '{' verbatim_balanced_text '}'
;

single_verbatim_balanced_text_character
: '<Any character except / (U+002F), @ (U+0040), \" (U+0022), $ (U+0024), ( (U+0028), ) (U+0029), [
(U+005B), ] (U+005D), { (U+007B) and } (U+007D)>'
| '</ (U+002F), if not directly followed by / (U+002F) or * (U+002A)>'
;

```

An *interpolated_string_literal* token is reinterpreted as multiple tokens and other input elements as follows, in

order of occurrence in the *interpolated_string_literal*.

- Occurrences of the following are reinterpreted as separate individual tokens: the leading `$` sign, *interpolated_regular_string_whole*, *interpolated_regular_string_start*, *interpolated_regular_string_mid*, *interpolated_regular_string_end*, *interpolated_verbatim_string_whole*, *interpolated_verbatim_string_start*, *interpolated_verbatim_string_mid* and *interpolated_verbatim_string_end*.
- Occurrences of *regular_balanced_text* and *verbatim_balanced_text* between these are reprocessed as an *input_section* ([Lexical analysis](#)) and are reinterpreted as the resulting sequence of input elements. These may in turn include interpolated string literal tokens to be reinterpreted.

Syntactic analysis will recombine the tokens into an *interpolated_string_expression* ([Interpolated strings](#)).

Examples TODO

The null literal

```
null_literal
: 'null'
;
```

The *null_literal* can be implicitly converted to a reference type or nullable type.

Operators and punctuators

There are several kinds of operators and punctuators. Operators are used in expressions to describe operations involving one or more operands. For example, the expression `a + b` uses the `+` operator to add the two operands `a` and `b`. Punctuators are for grouping and separating.

```
operator_or_punctuator
: '{' | '}' | '[' | ']' | '(' | ')' | '.' | ',' | ':' | ';'
| '+' | '-' | '*' | '/' | '%' | '&' | '|' | '^' | '!' | '~'
| '=' | '<' | '>' | '?' | '??' | '::' | '++' | '--' | '&&' | '||'
| '->' | '==' | '!=' | '<=' | '>=' | '+=' | '-=' | '*=' | '/=' | '%='
| '&=' | '|=' | '^=' | '<<' | '<<=' | '>='
;

right_shift
: '>>'
;

right_shift_assignment
: '>>='
;
```

The vertical bar in the *right_shift* and *right_shift_assignment* productions are used to indicate that, unlike other productions in the syntactic grammar, no characters of any kind (not even whitespace) are allowed between the tokens. These productions are treated specially in order to enable the correct handling of *type_parameter_lists* ([Type parameters](#)).

Pre-processing directives

The pre-processing directives provide the ability to conditionally skip sections of source files, to report error and warning conditions, and to delineate distinct regions of source code. The term "pre-processing directives" is used only for consistency with the C and C++ programming languages. In C#, there is no separate pre-processing step; pre-processing directives are processed as part of the lexical analysis phase.

```
pp_directive
: pp_declaration
| pp_conditional
| pp_line
| pp_diagnostic
| pp_region
| pp_pragma
;
```

The following pre-processing directives are available:

- `#define` and `#undef`, which are used to define and undefine, respectively, conditional compilation symbols ([Declaration directives](#)).
- `#if`, `#elif`, `#else`, and `#endif`, which are used to conditionally skip sections of source code ([Conditional compilation directives](#)).
- `#line`, which is used to control line numbers emitted for errors and warnings ([Line directives](#)).
- `#error` and `#warning`, which are used to issue errors and warnings, respectively ([Diagnostic directives](#)).
- `#region` and `#endregion`, which are used to explicitly mark sections of source code ([Region directives](#)).
- `#pragma`, which is used to specify optional contextual information to the compiler ([Pragma directives](#)).

A pre-processing directive always occupies a separate line of source code and always begins with a `#` character and a pre-processing directive name. White space may occur before the `#` character and between the `#` character and the directive name.

A source line containing a `#define`, `#undef`, `#if`, `#elif`, `#else`, `#endif`, `#line`, or `#endregion` directive may end with a single-line comment. Delimited comments (the `/* */` style of comments) are not permitted on source lines containing pre-processing directives.

Pre-processing directives are not tokens and are not part of the syntactic grammar of C#. However, pre-processing directives can be used to include or exclude sequences of tokens and can in that way affect the meaning of a C# program. For example, when compiled, the program:

```
#define A
#undef B

class C
{
    #if A
        void F() {}
    #else
        void G() {}
    #endif

    #if B
        void H() {}
    #else
        void I() {}
    #endif
}
```

results in the exact same sequence of tokens as the program:

```
class C
{
    void F() {}
    void I() {}
}
```

Thus, whereas lexically, the two programs are quite different, syntactically, they are identical.

Conditional compilation symbols

The conditional compilation functionality provided by the `#if`, `#elif`, `#else`, and `#endif` directives is controlled through pre-processing expressions ([Pre-processing expressions](#)) and conditional compilation symbols.

```
conditional_symbol
: '<Any identifier_or_keyword except true or false>'
;
```

A conditional compilation symbol has two possible states: ***defined*** or ***undefined***. At the beginning of the lexical processing of a source file, a conditional compilation symbol is undefined unless it has been explicitly defined by an external mechanism (such as a command-line compiler option). When a `#define` directive is processed, the conditional compilation symbol named in that directive becomes defined in that source file. The symbol remains defined until an `#undef` directive for that same symbol is processed, or until the end of the source file is reached. An implication of this is that `#define` and `#undef` directives in one source file have no effect on other source files in the same program.

When referenced in a pre-processing expression, a defined conditional compilation symbol has the boolean value `true`, and an undefined conditional compilation symbol has the boolean value `false`. There is no requirement that conditional compilation symbols be explicitly declared before they are referenced in pre-processing expressions. Instead, undeclared symbols are simply undefined and thus have the value `false`.

The name space for conditional compilation symbols is distinct and separate from all other named entities in a C# program. Conditional compilation symbols can only be referenced in `#define` and `#undef` directives and in pre-processing expressions.

Pre-processing expressions

Pre-processing expressions can occur in `#if` and `#elif` directives. The operators `!`, `==`, `!=`, `&&` and `||` are permitted in pre-processing expressions, and parentheses may be used for grouping.

```

pp_expression
: whitespace? pp_or_expression whitespace?
;

pp_or_expression
: pp_and_expression
| pp_or_expression whitespace? '||' whitespace? pp_and_expression
;

pp_and_expression
: pp_equality_expression
| pp_and_expression whitespace? '&&' whitespace? pp_equality_expression
;

pp_equality_expression
: pp_unary_expression
| pp_equality_expression whitespace? '==' whitespace? pp_unary_expression
| pp_equality_expression whitespace? '!=' whitespace? pp_unary_expression
;

pp_unary_expression
: pp_primary_expression
| '!' whitespace? pp_unary_expression
;

pp_primary_expression
: 'true'
| 'false'
| conditional_symbol
| '(' whitespace? pp_expression whitespace? ')'
;

```

When referenced in a pre-processing expression, a defined conditional compilation symbol has the boolean value `true`, and an undefined conditional compilation symbol has the boolean value `false`.

Evaluation of a pre-processing expression always yields a boolean value. The rules of evaluation for a pre-processing expression are the same as those for a constant expression ([Constant expressions](#)), except that the only user-defined entities that can be referenced are conditional compilation symbols.

Declaration directives

The declaration directives are used to define or undefine conditional compilation symbols.

```

pp_declaration
: whitespace? '#' whitespace? 'define' whitespace conditional_symbol pp_new_line
| whitespace? '#' whitespace? 'undef' whitespace conditional_symbol pp_new_line
;

pp_new_line
: whitespace? single_line_comment? new_line
;

```

The processing of a `#define` directive causes the given conditional compilation symbol to become defined, starting with the source line that follows the directive. Likewise, the processing of an `#undef` directive causes the given conditional compilation symbol to become undefined, starting with the source line that follows the directive.

Any `#define` and `#undef` directives in a source file must occur before the first *token* ([Tokens](#)) in the source file; otherwise a compile-time error occurs. In intuitive terms, `#define` and `#undef` directives must precede any "real code" in the source file.

The example:

```
#define Enterprise

#if Professional || Enterprise
    #define Advanced
#endif

namespace Megacorp.Data
{
    #if Advanced
    class PivotTable {...}
    #endif
}
```

is valid because the `#define` directives precede the first token (the `namespace` keyword) in the source file.

The following example results in a compile-time error because a `#define` follows real code:

```
#define A
namespace N
{
    #define B
    #if B
    class Class1 {}
    #endif
}
```

A `#define` may define a conditional compilation symbol that is already defined, without there being any intervening `#undef` for that symbol. The example below defines a conditional compilation symbol `A` and then defines it again.

```
#define A
#define A
```

A `#undef` may "undefine" a conditional compilation symbol that is not defined. The example below defines a conditional compilation symbol `A` and then undefines it twice; although the second `#undef` has no effect, it is still valid.

```
#define A
#undef A
#undef A
```

Conditional compilation directives

The conditional compilation directives are used to conditionally include or exclude portions of a source file.

```

pp_conditional
: pp_if_section pp_elif_section* pp_else_section? pp_endif
;

pp_if_section
: whitespace? '#' whitespace? 'if' whitespace pp_expression pp_new_line conditional_section?
;

pp_elif_section
: whitespace? '#' whitespace? 'elif' whitespace pp_expression pp_new_line conditional_section?
;

pp_else_section:
| whitespace? '#' whitespace? 'else' pp_new_line conditional_section?
;

pp_endif
: whitespace? '#' whitespace? 'endif' pp_new_line
;

conditional_section
: input_section
| skipped_section
;

skipped_section
: skipped_section_part+
;

skipped_section_part
: skipped_characters? new_line
| pp_directive
;

skipped_characters
: whitespace? not_number_sign input_character*
;

not_number_sign
: '<Any input_character except #>'
;

```

As indicated by the syntax, conditional compilation directives must be written as sets consisting of, in order, an `#if` directive, zero or more `#elif` directives, zero or one `#else` directive, and an `#endif` directive. Between the directives are conditional sections of source code. Each section is controlled by the immediately preceding directive. A conditional section may itself contain nested conditional compilation directives provided these directives form complete sets.

A *pp_conditional* selects at most one of the contained *conditional_sections* for normal lexical processing:

- The *pp_expressions* of the `#if` and `#elif` directives are evaluated in order until one yields `true`. If an expression yields `true`, the *conditional_section* of the corresponding directive is selected.
- If all *pp_expressions* yield `false`, and if an `#else` directive is present, the *conditional_section* of the `#else` directive is selected.
- Otherwise, no *conditional_section* is selected.

The selected *conditional_section*, if any, is processed as a normal *input_section*: the source code contained in the section must adhere to the lexical grammar; tokens are generated from the source code in the section; and pre-processing directives in the section have the prescribed effects.

The remaining *conditional_sections*, if any, are processed as *skipped_sections*: except for pre-processing directives, the source code in the section need not adhere to the lexical grammar; no tokens are generated from

the source code in the section; and pre-processing directives in the section must be lexically correct but are not otherwise processed. Within a *conditional_section* that is being processed as a *skipped_section*, any nested *conditional_sections* (contained in nested `#if ... #endif` and `#region ... #endregion` constructs) are also processed as *skipped_sections*.

The following example illustrates how conditional compilation directives can nest:

```
#define Debug      // Debugging on
#undef Trace      // Tracing off

class PurchaseTransaction
{
    void Commit() {
        #if Debug
            CheckConsistency();
            #if Trace
                WriteToLog(this.ToString());
            #endif
        #endif
        CommitHelper();
    }
}
```

Except for pre-processing directives, skipped source code is not subject to lexical analysis. For example, the following is valid despite the unterminated comment in the `#else` section:

```
#define Debug      // Debugging on

class PurchaseTransaction
{
    void Commit() {
        #if Debug
            CheckConsistency();
        #else
            /* Do something else
        #endif
    }
}
```

Note, however, that pre-processing directives are required to be lexically correct even in skipped sections of source code.

Pre-processing directives are not processed when they appear inside multi-line input elements. For example, the program:

```
class Hello
{
    static void Main() {
        System.Console.WriteLine(@"hello,
#if Debug
    world
#else
    Nebraska
#endif
    ");
    }
}
```

results in the output:

```
hello,
#ifdef Debug
    world
#else
    Nebraska
#endif
```

In peculiar cases, the set of pre-processing directives that is processed might depend on the evaluation of the *pp_expression*. The example:

```
#if X
    /*
#else
    /* */ class Q { }
#endif
```

always produces the same token stream (`class Q { }`), regardless of whether or not `X` is defined. If `X` is defined, the only processed directives are `#if` and `#endif`, due to the multi-line comment. If `X` is undefined, then three directives (`#if`, `#else`, `#endif`) are part of the directive set.

Diagnostic directives

The diagnostic directives are used to explicitly generate error and warning messages that are reported in the same way as other compile-time errors and warnings.

```
pp_diagnostic
: whitespace? '#' whitespace? 'error' pp_message
| whitespace? '#' whitespace? 'warning' pp_message
;

pp_message
: new_line
| whitespace input_character* new_line
;
```

The example:

```
#warning Code review needed before check-in

#ifdef Debug && Retail
    #error A build can't be both debug and retail
#endif

class Test {...}
```

always produces a warning ("Code review needed before check-in"), and produces a compile-time error ("A build can't be both debug and retail") if the conditional symbols `Debug` and `Retail` are both defined. Note that a *pp_message* can contain arbitrary text; specifically, it need not contain well-formed tokens, as shown by the single quote in the word `can't`.

Region directives

The region directives are used to explicitly mark regions of source code.


```

pp_region
: pp_start_region conditional_section? pp_end_region
;

pp_start_region
: whitespace? '#' whitespace? 'region' pp_message
;

pp_end_region
: whitespace? '#' whitespace? 'endregion' pp_message
;

```

No semantic meaning is attached to a region; regions are intended for use by the programmer or by automated tools to mark a section of source code. The message specified in a `#region` or `#endregion` directive likewise has no semantic meaning; it merely serves to identify the region. Matching `#region` and `#endregion` directives may have different *pp_messages*.

The lexical processing of a region:

```

#region
...
#endregion

```

corresponds exactly to the lexical processing of a conditional compilation directive of the form:

```

#if true
...
#endif

```

Line directives

Line directives may be used to alter the line numbers and source file names that are reported by the compiler in output such as warnings and errors, and that are used by caller info attributes ([Caller info attributes](#)).

Line directives are most commonly used in meta-programming tools that generate C# source code from some other text input.

```

pp_line
: whitespace? '#' whitespace? 'line' whitespace line_indicator pp_new_line
;

line_indicator
: decimal_digit+ whitespace file_name
| decimal_digit+
| 'default'
| 'hidden'
;

file_name
: ''' file_name_character+ '''
;

file_name_character
: '<Any input_character except ">'
;

```

When no `#line` directives are present, the compiler reports true line numbers and source file names in its output. When processing a `#line` directive that includes a *line_indicator* that is not `default`, the compiler treats the line after the directive as having the given line number (and file name, if specified).

A `#line default` directive reverses the effect of all preceding `#line` directives. The compiler reports true line information for subsequent lines, precisely as if no `#line` directives had been processed.

A `#line hidden` directive has no effect on the file and line numbers reported in error messages, but does affect source level debugging. When debugging, all lines between a `#line hidden` directive and the subsequent `#line` directive (that is not `#line hidden`) have no line number information. When stepping through code in the debugger, these lines will be skipped entirely.

Note that a *file_name* differs from a regular string literal in that escape characters are not processed; the `"\"` character simply designates an ordinary backslash character within a *file_name*.

Pragma directives

The `#pragma` preprocessing directive is used to specify optional contextual information to the compiler. The information supplied in a `#pragma` directive will never change program semantics.

```
pp_pragma
    : whitespace? '#' whitespace? 'pragma' whitespace pragma_body pp_new_line
    ;

pragma_body
    : pragma_warning_body
    ;
```

C# provides `#pragma` directives to control compiler warnings. Future versions of the language may include additional `#pragma` directives. To ensure interoperability with other C# compilers, the Microsoft C# compiler does not issue compilation errors for unknown `#pragma` directives; such directives do however generate warnings.

Pragma warning

The `#pragma warning` directive is used to disable or restore all or a particular set of warning messages during compilation of the subsequent program text.

```
pragma_warning_body
    : 'warning' whitespace warning_action
    | 'warning' whitespace warning_action whitespace warning_list
    ;

warning_action
    : 'disable'
    | 'restore'
    ;

warning_list
    : decimal_digit+ (whitespace? ',' whitespace? decimal_digit+)*
    ;
```

A `#pragma warning` directive that omits the warning list affects all warnings. A `#pragma warning` directive that includes a warning list affects only those warnings that are specified in the list.

A `#pragma warning disable` directive disables all or the given set of warnings.

A `#pragma warning restore` directive restores all or the given set of warnings to the state that was in effect at the beginning of the compilation unit. Note that if a particular warning was disabled externally, a `#pragma warning restore` (whether for all or the specific warning) will not re-enable that warning.

The following example shows use of `#pragma warning` to temporarily disable the warning reported when obsoleted members are referenced, using the warning number from the Microsoft C# compiler.

```
using System;

class Program
{
    [Obsolete]
    static void Foo() {}

    static void Main() {
#pragma warning disable 612
        Foo();
#pragma warning restore 612
    }
}
```

Basic concepts

12/28/2021 • 48 minutes to read • [Edit Online](#)

Application Startup

An assembly that has an *entry point* is called an *application*. When an application is run, a new *application domain* is created. Several different instantiations of an application may exist on the same machine at the same time, and each has its own application domain.

An application domain enables application isolation by acting as a container for application state. An application domain acts as a container and boundary for the types defined in the application and the class libraries it uses. Types loaded into one application domain are distinct from the same type loaded into another application domain, and instances of objects are not directly shared between application domains. For instance, each application domain has its own copy of static variables for these types, and a static constructor for a type is run at most once per application domain. Implementations are free to provide implementation-specific policy or mechanisms for the creation and destruction of application domains.

Application startup occurs when the execution environment calls a designated method, which is referred to as the application's entry point. This entry point method is always named `Main`, and can have one of the following signatures:

```
static void Main() {...}

static void Main(string[] args) {...}

static int Main() {...}

static int Main(string[] args) {...}
```

As shown, the entry point may optionally return an `int` value. This return value is used in application termination ([Application termination](#)).

The entry point may optionally have one formal parameter. The parameter may have any name, but the type of the parameter must be `string[]`. If the formal parameter is present, the execution environment creates and passes a `string[]` argument containing the command-line arguments that were specified when the application was started. The `string[]` argument is never null, but it may have a length of zero if no command-line arguments were specified.

Since C# supports method overloading, a class or struct may contain multiple definitions of some method, provided each has a different signature. However, within a single program, no class or struct may contain more than one method called `Main` whose definition qualifies it to be used as an application entry point. Other overloaded versions of `Main` are permitted, however, provided they have more than one parameter, or their only parameter is other than type `string[]`.

An application can be made up of multiple classes or structs. It is possible for more than one of these classes or structs to contain a method called `Main` whose definition qualifies it to be used as an application entry point. In such cases, an external mechanism (such as a command-line compiler option) must be used to select one of these `Main` methods as the entry point.

In C#, every method must be defined as a member of a class or struct. Ordinarily, the declared accessibility ([Declared accessibility](#)) of a method is determined by the access modifiers ([Access modifiers](#)) specified in its declaration, and similarly the declared accessibility of a type is determined by the access modifiers specified in

its declaration. In order for a given method of a given type to be callable, both the type and the member must be accessible. However, the application entry point is a special case. Specifically, the execution environment can access the application's entry point regardless of its declared accessibility and regardless of the declared accessibility of its enclosing type declarations.

The application entry point method may not be in a generic class declaration.

In all other respects, entry point methods behave like those that are not entry points.

Application termination

Application termination returns control to the execution environment.

If the return type of the application's *entry point* method is `int`, the value returned serves as the application's *termination status code*. The purpose of this code is to allow communication of success or failure to the execution environment.

If the return type of the entry point method is `void`, reaching the right brace (`}`) which terminates that method, or executing a `return` statement that has no expression, results in a termination status code of `0`.

Prior to an application's termination, destructors for all of its objects that have not yet been garbage collected are called, unless such cleanup has been suppressed (by a call to the library method `GC.SuppressFinalize`, for example).

Declarations

Declarations in a C# program define the constituent elements of the program. C# programs are organized using namespaces ([Namespaces](#)), which can contain type declarations and nested namespace declarations. Type declarations ([Type declarations](#)) are used to define classes ([Classes](#)), structs ([Structs](#)), interfaces ([Interfaces](#)), enums ([Enums](#)), and delegates ([Delegates](#)). The kinds of members permitted in a type declaration depend on the form of the type declaration. For instance, class declarations can contain declarations for constants ([Constants](#)), fields ([Fields](#)), methods ([Methods](#)), properties ([Properties](#)), events ([Events](#)), indexers ([Indexers](#)), operators ([Operators](#)), instance constructors ([Instance constructors](#)), static constructors ([Static constructors](#)), destructors ([Destructors](#)), and nested types ([Nested types](#)).

A declaration defines a name in the *declaration space* to which the declaration belongs. Except for overloaded members ([Signatures and overloading](#)), it is a compile-time error to have two or more declarations that introduce members with the same name in a declaration space. It is never possible for a declaration space to contain different kinds of members with the same name. For example, a declaration space can never contain a field and a method by the same name.

There are several different types of declaration spaces, as described in the following.

- Within all source files of a program, *namespace_member_declarations* with no enclosing *namespace_declaration* are members of a single combined declaration space called the *global declaration space*.
- Within all source files of a program, *namespace_member_declarations* within *namespace_declarations* that have the same fully qualified namespace name are members of a single combined declaration space.
- Each class, struct, or interface declaration creates a new declaration space. Names are introduced into this declaration space through *class_member_declarations*, *struct_member_declarations*, *interface_member_declarations*, or *type_parameters*. Except for overloaded instance constructor declarations and static constructor declarations, a class or struct cannot contain a member declaration with the same name as the class or struct. A class, struct, or interface permits the declaration of overloaded methods and indexers. Furthermore, a class or struct permits the declaration of overloaded instance constructors and operators. For example, a class, struct, or interface may contain multiple method declarations with the same

name, provided these method declarations differ in their signature ([Signatures and overloading](#)). Note that base classes do not contribute to the declaration space of a class, and base interfaces do not contribute to the declaration space of an interface. Thus, a derived class or interface is allowed to declare a member with the same name as an inherited member. Such a member is said to *hide* the inherited member.

- Each delegate declaration creates a new declaration space. Names are introduced into this declaration space through formal parameters (*fixed_parameters* and *parameter_arrays*) and *type_parameters*.
- Each enumeration declaration creates a new declaration space. Names are introduced into this declaration space through *enum_member_declarations*.
- Each method declaration, indexer declaration, operator declaration, instance constructor declaration and anonymous function creates a new declaration space called a **local variable declaration space**. Names are introduced into this declaration space through formal parameters (*fixed_parameters* and *parameter_arrays*) and *type_parameters*. The body of the function member or anonymous function, if any, is considered to be nested within the local variable declaration space. It is an error for a local variable declaration space and a nested local variable declaration space to contain elements with the same name. Thus, within a nested declaration space it is not possible to declare a local variable or constant with the same name as a local variable or constant in an enclosing declaration space. It is possible for two declaration spaces to contain elements with the same name as long as neither declaration space contains the other.
- Each *block* or *switch_block*, as well as a *for*, *foreach* and *using* statement, creates a local variable declaration space for local variables and local constants. Names are introduced into this declaration space through *local_variable_declarations* and *local_constant_declarations*. Note that blocks that occur as or within the body of a function member or anonymous function are nested within the local variable declaration space declared by those functions for their parameters. Thus it is an error to have e.g. a method with a local variable and a parameter of the same name.
- Each *block* or *switch_block* creates a separate declaration space for labels. Names are introduced into this declaration space through *labeled_statements*, and the names are referenced through *goto_statements*. The **label declaration space** of a block includes any nested blocks. Thus, within a nested block it is not possible to declare a label with the same name as a label in an enclosing block.

The textual order in which names are declared is generally of no significance. In particular, textual order is not significant for the declaration and use of namespaces, constants, methods, properties, events, indexers, operators, instance constructors, destructors, static constructors, and types. Declaration order is significant in the following ways:

- Declaration order for field declarations and local variable declarations determines the order in which their initializers (if any) are executed.
- Local variables must be defined before they are used ([Scopes](#)).
- Declaration order for enum member declarations ([Enum members](#)) is significant when *constant_expression* values are omitted.

The declaration space of a namespace is "open ended", and two namespace declarations with the same fully qualified name contribute to the same declaration space. For example

```

namespace Megacorp.Data
{
    class Customer
    {
        ...
    }
}

namespace Megacorp.Data
{
    class Order
    {
        ...
    }
}

```

The two namespace declarations above contribute to the same declaration space, in this case declaring two classes with the fully qualified names `Megacorp.Data.Customer` and `Megacorp.Data.Order`. Because the two declarations contribute to the same declaration space, it would have caused a compile-time error if each contained a declaration of a class with the same name.

As specified above, the declaration space of a block includes any nested blocks. Thus, in the following example, the `F` and `G` methods result in a compile-time error because the name `i` is declared in the outer block and cannot be redeclared in the inner block. However, the `H` and `I` methods are valid since the two `i`'s are declared in separate non-nested blocks.

```

class A
{
    void F() {
        int i = 0;
        if (true) {
            int i = 1;
        }
    }

    void G() {
        if (true) {
            int i = 0;
        }
        int i = 1;
    }

    void H() {
        if (true) {
            int i = 0;
        }
        if (true) {
            int i = 1;
        }
    }

    void I() {
        for (int i = 0; i < 10; i++)
            H();
        for (int i = 0; i < 10; i++)
            H();
    }
}

```

Members

Namespaces and types have **members**. The members of an entity are generally available through the use of a qualified name that starts with a reference to the entity, followed by a "." token, followed by the name of the member.

Members of a type are either declared in the type declaration or **inherited** from the base class of the type. When a type inherits from a base class, all members of the base class, except instance constructors, destructors and static constructors, become members of the derived type. The declared accessibility of a base class member does not control whether the member is inherited—inheritance extends to any member that isn't an instance constructor, static constructor, or destructor. However, an inherited member may not be accessible in a derived type, either because of its declared accessibility ([Declared accessibility](#)) or because it is hidden by a declaration in the type itself ([Hiding through inheritance](#)).

Namespace members

Namespaces and types that have no enclosing namespace are members of the **global namespace**. This corresponds directly to the names declared in the global declaration space.

Namespaces and types declared within a namespace are members of that namespace. This corresponds directly to the names declared in the declaration space of the namespace.

Namespaces have no access restrictions. It is not possible to declare private, protected, or internal namespaces, and namespace names are always publicly accessible.

Struct members

The members of a struct are the members declared in the struct and the members inherited from the struct's direct base class `System.ValueType` and the indirect base class `object`.

The members of a simple type correspond directly to the members of the struct type aliased by the simple type:

- The members of `sbyte` are the members of the `System.SByte` struct.
- The members of `byte` are the members of the `System.Byte` struct.
- The members of `short` are the members of the `System.Int16` struct.
- The members of `ushort` are the members of the `System.UInt16` struct.
- The members of `int` are the members of the `System.Int32` struct.
- The members of `uint` are the members of the `System.UInt32` struct.
- The members of `long` are the members of the `System.Int64` struct.
- The members of `ulong` are the members of the `System.UInt64` struct.
- The members of `char` are the members of the `System.Char` struct.
- The members of `float` are the members of the `System.Single` struct.
- The members of `double` are the members of the `System.Double` struct.
- The members of `decimal` are the members of the `System.Decimal` struct.
- The members of `bool` are the members of the `System.Boolean` struct.

Enumeration members

The members of an enumeration are the constants declared in the enumeration and the members inherited from the enumeration's direct base class `System.Enum` and the indirect base classes `System.ValueType` and `object`.

Class members

The members of a class are the members declared in the class and the members inherited from the base class (except for class `object` which has no base class). The members inherited from the base class include the constants, fields, methods, properties, events, indexers, operators, and types of the base class, but not the instance constructors, destructors and static constructors of the base class. Base class members are inherited without regard to their accessibility.

A class declaration may contain declarations of constants, fields, methods, properties, events, indexers, operators, instance constructors, destructors, static constructors and types.

The members of `object` and `string` correspond directly to the members of the class types they alias:

- The members of `object` are the members of the `System.Object` class.
- The members of `string` are the members of the `System.String` class.

Interface members

The members of an interface are the members declared in the interface and in all base interfaces of the interface.

The members in class `object` are not, strictly speaking, members of any interface ([Interface members](#)).

However, the members in class `object` are available via member lookup in any interface type ([Member lookup](#)).

Array members

The members of an array are the members inherited from class `System.Array`.

Delegate members

The members of a delegate are the members inherited from class `System.Delegate`.

Member access

Declarations of members allow control over member access. The accessibility of a member is established by the declared accessibility ([Declared accessibility](#)) of the member combined with the accessibility of the immediately containing type, if any.

When access to a particular member is allowed, the member is said to be *accessible*. Conversely, when access to a particular member is disallowed, the member is said to be *inaccessible*. Access to a member is permitted when the textual location in which the access takes place is included in the accessibility domain ([Accessibility domains](#)) of the member.

Declared accessibility

The *declared accessibility* of a member can be one of the following:

- Public, which is selected by including a `public` modifier in the member declaration. The intuitive meaning of `public` is "access not limited".
- Protected, which is selected by including a `protected` modifier in the member declaration. The intuitive meaning of `protected` is "access limited to the containing class or types derived from the containing class".
- Internal, which is selected by including an `internal` modifier in the member declaration. The intuitive meaning of `internal` is "access limited to this program".
- Protected internal (meaning protected or internal), which is selected by including both a `protected` and an `internal` modifier in the member declaration. The intuitive meaning of `protected internal` is "access limited to this program or types derived from the containing class".
- Private, which is selected by including a `private` modifier in the member declaration. The intuitive meaning of `private` is "access limited to the containing type".

Depending on the context in which a member declaration takes place, only certain types of declared accessibility are permitted. Furthermore, when a member declaration does not include any access modifiers, the context in which the declaration takes place determines the default declared accessibility.

- Namespaces implicitly have `public` declared accessibility. No access modifiers are allowed on namespace declarations.
- Types declared in compilation units or namespaces can have `public` or `internal` declared accessibility and default to `internal` declared accessibility.
- Class members can have any of the five kinds of declared accessibility and default to `private` declared

accessibility. (Note that a type declared as a member of a class can have any of the five kinds of declared accessibility, whereas a type declared as a member of a namespace can have only `public` or `internal` declared accessibility.)

- Struct members can have `public`, `internal`, or `private` declared accessibility and default to `private` declared accessibility because structs are implicitly sealed. Struct members introduced in a struct (that is, not inherited by that struct) cannot have `protected` or `protected internal` declared accessibility. (Note that a type declared as a member of a struct can have `public`, `internal`, or `private` declared accessibility, whereas a type declared as a member of a namespace can have only `public` or `internal` declared accessibility.)
- Interface members implicitly have `public` declared accessibility. No access modifiers are allowed on interface member declarations.
- Enumeration members implicitly have `public` declared accessibility. No access modifiers are allowed on enumeration member declarations.

Accessibility domains

The **accessibility domain** of a member consists of the (possibly disjoint) sections of program text in which access to the member is permitted. For purposes of defining the accessibility domain of a member, a member is said to be **top-level** if it is not declared within a type, and a member is said to be **nested** if it is declared within another type. Furthermore, the **program text** of a program is defined as all program text contained in all source files of the program, and the program text of a type is defined as all program text contained in the *type_declarations* of that type (including, possibly, types that are nested within the type).

The accessibility domain of a predefined type (such as `object`, `int`, or `double`) is unlimited.

The accessibility domain of a top-level unbound type `T` ([Bound and unbound types](#)) that is declared in a program `P` is defined as follows:

- If the declared accessibility of `T` is `public`, the accessibility domain of `T` is the program text of `P` and any program that references `P`.
- If the declared accessibility of `T` is `internal`, the accessibility domain of `T` is the program text of `P`.

From these definitions it follows that the accessibility domain of a top-level unbound type is always at least the program text of the program in which that type is declared.

The accessibility domain for a constructed type `T<A1, ..., An>` is the intersection of the accessibility domain of the unbound generic type `T` and the accessibility domains of the type arguments `A1, ..., An`.

The accessibility domain of a nested member `M` declared in a type `T` within a program `P` is defined as follows (noting that `M` itself may possibly be a type):

- If the declared accessibility of `M` is `public`, the accessibility domain of `M` is the accessibility domain of `T`.
- If the declared accessibility of `M` is `protected internal`, let `D` be the union of the program text of `P` and the program text of any type derived from `T`, which is declared outside `P`. The accessibility domain of `M` is the intersection of the accessibility domain of `T` with `D`.
- If the declared accessibility of `M` is `protected`, let `D` be the union of the program text of `T` and the program text of any type derived from `T`. The accessibility domain of `M` is the intersection of the accessibility domain of `T` with `D`.
- If the declared accessibility of `M` is `internal`, the accessibility domain of `M` is the intersection of the accessibility domain of `T` with the program text of `P`.
- If the declared accessibility of `M` is `private`, the accessibility domain of `M` is the program text of `T`.

From these definitions it follows that the accessibility domain of a nested member is always at least the program text of the type in which the member is declared. Furthermore, it follows that the accessibility domain of a member is never more inclusive than the accessibility domain of the type in which the member is declared.

In intuitive terms, when a type or member `M` is accessed, the following steps are evaluated to ensure that the access is permitted:

- First, if `M` is declared within a type (as opposed to a compilation unit or a namespace), a compile-time error occurs if that type is not accessible.
- Then, if `M` is `public`, the access is permitted.
- Otherwise, if `M` is `protected internal`, the access is permitted if it occurs within the program in which `M` is declared, or if it occurs within a class derived from the class in which `M` is declared and takes place through the derived class type ([Protected access for instance members](#)).
- Otherwise, if `M` is `protected`, the access is permitted if it occurs within the class in which `M` is declared, or if it occurs within a class derived from the class in which `M` is declared and takes place through the derived class type ([Protected access for instance members](#)).
- Otherwise, if `M` is `internal`, the access is permitted if it occurs within the program in which `M` is declared.
- Otherwise, if `M` is `private`, the access is permitted if it occurs within the type in which `M` is declared.
- Otherwise, the type or member is inaccessible, and a compile-time error occurs.

In the example

```
public class A
{
    public static int X;
    internal static int Y;
    private static int Z;
}

internal class B
{
    public static int X;
    internal static int Y;
    private static int Z;

    public class C
    {
        public static int X;
        internal static int Y;
        private static int Z;
    }

    private class D
    {
        public static int X;
        internal static int Y;
        private static int Z;
    }
}
```

the classes and members have the following accessibility domains:

- The accessibility domain of `A` and `A.X` is unlimited.
- The accessibility domain of `A.Y`, `B`, `B.X`, `B.Y`, `B.C`, `B.C.X`, and `B.C.Y` is the program text of the containing program.
- The accessibility domain of `A.Z` is the program text of `A`.
- The accessibility domain of `B.Z` and `B.D` is the program text of `B`, including the program text of `B.C` and `B.D`.
- The accessibility domain of `B.C.Z` is the program text of `B.C`.
- The accessibility domain of `B.D.X` and `B.D.Y` is the program text of `B`, including the program text of `B.C`

and `B.D`.

- The accessibility domain of `B.D.Z` is the program text of `B.D`.

As the example illustrates, the accessibility domain of a member is never larger than that of a containing type. For example, even though all `X` members have public declared accessibility, all but `A.X` have accessibility domains that are constrained by a containing type.

As described in [Members](#), all members of a base class, except for instance constructors, destructors and static constructors, are inherited by derived types. This includes even private members of a base class. However, the accessibility domain of a private member includes only the program text of the type in which the member is declared. In the example

```
class A
{
    int x;

    static void F(B b) {
        b.x = 1;      // Ok
    }
}

class B: A
{
    static void F(B b) {
        b.x = 1;      // Error, x not accessible
    }
}
```

the `B` class inherits the private member `x` from the `A` class. Because the member is private, it is only accessible within the *class_body* of `A`. Thus, the access to `b.x` succeeds in the `A.F` method, but fails in the `B.F` method.

Protected access for instance members

When a `protected` instance member is accessed outside the program text of the class in which it is declared, and when a `protected internal` instance member is accessed outside the program text of the program in which it is declared, the access must take place within a class declaration that derives from the class in which it is declared. Furthermore, the access is required to take place through an instance of that derived class type or a class type constructed from it. This restriction prevents one derived class from accessing protected members of other derived classes, even when the members are inherited from the same base class.

Let `B` be a base class that declares a protected instance member `M`, and let `D` be a class that derives from `B`. Within the *class_body* of `D`, access to `M` can take one of the following forms:

- An unqualified *type_name* or *primary_expression* of the form `M`.
- A *primary_expression* of the form `E.M`, provided the type of `E` is `T` or a class derived from `T`, where `T` is the class type `D`, or a class type constructed from `D`.
- A *primary_expression* of the form `base.M`.

In addition to these forms of access, a derived class can access a protected instance constructor of a base class in a *constructor_initializer* ([Constructor initializers](#)).

In the example

```

public class A
{
    protected int x;

    static void F(A a, B b) {
        a.x = 1;        // Ok
        b.x = 1;        // Ok
    }
}

public class B: A
{
    static void F(A a, B b) {
        a.x = 1;        // Error, must access through instance of B
        b.x = 1;        // Ok
    }
}

```

within `A`, it is possible to access `x` through instances of both `A` and `B`, since in either case the access takes place through an instance of `A` or a class derived from `A`. However, within `B`, it is not possible to access `x` through an instance of `A`, since `A` does not derive from `B`.

In the example

```

class C<T>
{
    protected T x;
}

class D<T>: C<T>
{
    static void F() {
        D<T> dt = new D<T>();
        D<int> di = new D<int>();
        D<string> ds = new D<string>();
        dt.x = default(T);
        di.x = 123;
        ds.x = "test";
    }
}

```

the three assignments to `x` are permitted because they all take place through instances of class types constructed from the generic type.

Accessibility constraints

Several constructs in the C# language require a type to be *at least as accessible as* a member or another type. A type `T` is said to be at least as accessible as a member or type `M` if the accessibility domain of `T` is a superset of the accessibility domain of `M`. In other words, `T` is at least as accessible as `M` if `T` is accessible in all contexts in which `M` is accessible.

The following accessibility constraints exist:

- The direct base class of a class type must be at least as accessible as the class type itself.
- The explicit base interfaces of an interface type must be at least as accessible as the interface type itself.
- The return type and parameter types of a delegate type must be at least as accessible as the delegate type itself.
- The type of a constant must be at least as accessible as the constant itself.
- The type of a field must be at least as accessible as the field itself.
- The return type and parameter types of a method must be at least as accessible as the method itself.

- The type of a property must be at least as accessible as the property itself.
- The type of an event must be at least as accessible as the event itself.
- The type and parameter types of an indexer must be at least as accessible as the indexer itself.
- The return type and parameter types of an operator must be at least as accessible as the operator itself.
- The parameter types of an instance constructor must be at least as accessible as the instance constructor itself.

In the example

```
class A {...}

public class B: A {...}
```

the `B` class results in a compile-time error because `A` is not at least as accessible as `B`.

Likewise, in the example

```
class A {...}

public class B
{
    A F() {...}

    internal A G() {...}

    public A H() {...}
}
```

the `H` method in `B` results in a compile-time error because the return type `A` is not at least as accessible as the method.

Signatures and overloading

Methods, instance constructors, indexers, and operators are characterized by their *signatures*:

- The signature of a method consists of the name of the method, the number of type parameters and the type and kind (value, reference, or output) of each of its formal parameters, considered in the order left to right. For these purposes, any type parameter of the method that occurs in the type of a formal parameter is identified not by its name, but by its ordinal position in the type argument list of the method. The signature of a method specifically does not include the return type, the `params` modifier that may be specified for the right-most parameter, nor the optional type parameter constraints.
- The signature of an instance constructor consists of the type and kind (value, reference, or output) of each of its formal parameters, considered in the order left to right. The signature of an instance constructor specifically does not include the `params` modifier that may be specified for the right-most parameter.
- The signature of an indexer consists of the type of each of its formal parameters, considered in the order left to right. The signature of an indexer specifically does not include the element type, nor does it include the `params` modifier that may be specified for the right-most parameter.
- The signature of an operator consists of the name of the operator and the type of each of its formal parameters, considered in the order left to right. The signature of an operator specifically does not include the result type.

Signatures are the enabling mechanism for *overloading* of members in classes, structs, and interfaces:

- Overloading of methods permits a class, struct, or interface to declare multiple methods with the same name, provided their signatures are unique within that class, struct, or interface.

- Overloading of instance constructors permits a class or struct to declare multiple instance constructors, provided their signatures are unique within that class or struct.
- Overloading of indexers permits a class, struct, or interface to declare multiple indexers, provided their signatures are unique within that class, struct, or interface.
- Overloading of operators permits a class or struct to declare multiple operators with the same name, provided their signatures are unique within that class or struct.

Although `out` and `ref` parameter modifiers are considered part of a signature, members declared in a single type cannot differ in signature solely by `ref` and `out`. A compile-time error occurs if two members are declared in the same type with signatures that would be the same if all parameters in both methods with `out` modifiers were changed to `ref` modifiers. For other purposes of signature matching (e.g., hiding or overriding), `ref` and `out` are considered part of the signature and do not match each other. (This restriction is to allow C# programs to be easily translated to run on the Common Language Infrastructure (CLI), which does not provide a way to define methods that differ solely in `ref` and `out`.)

For the purposes of signatures, the types `object` and `dynamic` are considered the same. Members declared in a single type can therefore not differ in signature solely by `object` and `dynamic`.

The following example shows a set of overloaded method declarations along with their signatures.

```
interface ITest
{
    void F();                // F()

    void F(int x);           // F(int)

    void F(ref int x);        // F(ref int)

    void F(out int x);        // F(out int)    error

    void F(int x, int y);     // F(int, int)

    int F(string s);         // F(string)

    int F(int x);            // F(int)        error

    void F(string[] a);       // F(string[])

    void F(params string[] a); // F(string[])    error
}
```

Note that any `ref` and `out` parameter modifiers ([Method parameters](#)) are part of a signature. Thus, `F(int)` and `F(ref int)` are unique signatures. However, `F(ref int)` and `F(out int)` cannot be declared within the same interface because their signatures differ solely by `ref` and `out`. Also, note that the return type and the `params` modifier are not part of a signature, so it is not possible to overload solely based on return type or on the inclusion or exclusion of the `params` modifier. As such, the declarations of the methods `F(int)` and `F(params string[])` identified above result in a compile-time error.

Scopes

The *scope* of a name is the region of program text within which it is possible to refer to the entity declared by the name without qualification of the name. Scopes can be *nested*, and an inner scope may redeclare the meaning of a name from an outer scope (this does not, however, remove the restriction imposed by [Declarations](#) that within a nested block it is not possible to declare a local variable with the same name as a local variable in an enclosing block). The name from the outer scope is then said to be *hidden* in the region of program text covered by the inner scope, and access to the outer name is only possible by qualifying the name.

- The scope of a namespace member declared by a *namespace_member_declaration* ([Namespace members](#)) with no enclosing *namespace_declaration* is the entire program text.
- The scope of a namespace member declared by a *namespace_member_declaration* within a *namespace_declaration* whose fully qualified name is `N` is the *namespace_body* of every *namespace_declaration* whose fully qualified name is `N` or starts with `N`, followed by a period.
- The scope of name defined by an *extern_alias_directive* extends over the *using_directives*, *global_attributes* and *namespace_member_declarations* of its immediately containing compilation unit or namespace body. An *extern_alias_directive* does not contribute any new members to the underlying declaration space. In other words, an *extern_alias_directive* is not transitive, but, rather, affects only the compilation unit or namespace body in which it occurs.
- The scope of a name defined or imported by a *using_directive* ([Using directives](#)) extends over the *namespace_member_declarations* of the *compilation_unit* or *namespace_body* in which the *using_directive* occurs. A *using_directive* may make zero or more namespace, type or member names available within a particular *compilation_unit* or *namespace_body*, but does not contribute any new members to the underlying declaration space. In other words, a *using_directive* is not transitive but rather affects only the *compilation_unit* or *namespace_body* in which it occurs.
- The scope of a type parameter declared by a *type_parameter_list* on a *class_declaration* ([Class declarations](#)) is the *class_base*, *type_parameter_constraints_clauses*, and *class_body* of that *class_declaration*.
- The scope of a type parameter declared by a *type_parameter_list* on a *struct_declaration* ([Struct declarations](#)) is the *struct_interfaces*, *type_parameter_constraints_clauses*, and *struct_body* of that *struct_declaration*.
- The scope of a type parameter declared by a *type_parameter_list* on an *interface_declaration* ([Interface declarations](#)) is the *interface_base*, *type_parameter_constraints_clauses*, and *interface_body* of that *interface_declaration*.
- The scope of a type parameter declared by a *type_parameter_list* on a *delegate_declaration* ([Delegate declarations](#)) is the *return_type*, *formal_parameter_list*, and *type_parameter_constraints_clauses* of that *delegate_declaration*.
- The scope of a member declared by a *class_member_declaration* ([Class body](#)) is the *class_body* in which the declaration occurs. In addition, the scope of a class member extends to the *class_body* of those derived classes that are included in the accessibility domain ([Accessibility domains](#)) of the member.
- The scope of a member declared by a *struct_member_declaration* ([Struct members](#)) is the *struct_body* in which the declaration occurs.
- The scope of a member declared by an *enum_member_declaration* ([Enum members](#)) is the *enum_body* in which the declaration occurs.
- The scope of a parameter declared in a *method_declaration* ([Methods](#)) is the *method_body* of that *method_declaration*.
- The scope of a parameter declared in an *indexer_declaration* ([Indexers](#)) is the *accessor_declarations* of that *indexer_declaration*.
- The scope of a parameter declared in an *operator_declaration* ([Operators](#)) is the *block* of that *operator_declaration*.
- The scope of a parameter declared in a *constructor_declaration* ([Instance constructors](#)) is the *constructor_initializer* and *block* of that *constructor_declaration*.
- The scope of a parameter declared in a *lambda_expression* ([Anonymous function expressions](#)) is the *anonymous_function_body* of that *lambda_expression*.
- The scope of a parameter declared in an *anonymous_method_expression* ([Anonymous function expressions](#)) is the *block* of that *anonymous_method_expression*.
- The scope of a label declared in a *labeled_statement* ([Labeled statements](#)) is the *block* in which the declaration occurs.
- The scope of a local variable declared in a *local_variable_declaration* ([Local variable declarations](#)) is the block in which the declaration occurs.

- The scope of a local variable declared in a *switch_block* of a `switch` statement ([The switch statement](#)) is the *switch_block*.
- The scope of a local variable declared in a *for_initializer* of a `for` statement ([The for statement](#)) is the *for_initializer*, the *for_condition*, the *for_iterator*, and the contained *statement* of the `for` statement.
- The scope of a local constant declared in a *local_constant_declaration* ([Local constant declarations](#)) is the block in which the declaration occurs. It is a compile-time error to refer to a local constant in a textual position that precedes its *constant_declarator*.
- The scope of a variable declared as part of a *foreach_statement*, *using_statement*, *lock_statement* or *query_expression* is determined by the expansion of the given construct.

Within the scope of a namespace, class, struct, or enumeration member it is possible to refer to the member in a textual position that precedes the declaration of the member. For example

```
class A
{
    void F() {
        i = 1;
    }

    int i = 0;
}
```

Here, it is valid for `F` to refer to `i` before it is declared.

Within the scope of a local variable, it is a compile-time error to refer to the local variable in a textual position that precedes the *local_variable_declarator* of the local variable. For example

```
class A
{
    int i = 0;

    void F() {
        i = 1;           // Error, use precedes declaration
        int i;
        i = 2;
    }

    void G() {
        int j = (j = 1); // Valid
    }

    void H() {
        int a = 1, b = ++a; // Valid
    }
}
```

In the `F` method above, the first assignment to `i` specifically does not refer to the field declared in the outer scope. Rather, it refers to the local variable and it results in a compile-time error because it textually precedes the declaration of the variable. In the `G` method, the use of `j` in the initializer for the declaration of `j` is valid because the use does not precede the *local_variable_declarator*. In the `H` method, a subsequent *local_variable_declarator* correctly refers to a local variable declared in an earlier *local_variable_declarator* within the same *local_variable_declaration*.

The scoping rules for local variables are designed to guarantee that the meaning of a name used in an expression context is always the same within a block. If the scope of a local variable were to extend only from its declaration to the end of the block, then in the example above, the first assignment would assign to the instance variable and the second assignment would assign to the local variable, possibly leading to compile-time errors if

the statements of the block were later to be rearranged.

The meaning of a name within a block may differ based on the context in which the name is used. In the example

```
using System;

class A {}

class Test
{
    static void Main() {
        string A = "hello, world";
        string s = A;                                // expression context

        Type t = typeof(A);                          // type context

        Console.WriteLine(s);                        // writes "hello, world"
        Console.WriteLine(t);                        // writes "A"
    }
}
```

the name `A` is used in an expression context to refer to the local variable `A` and in a type context to refer to the class `A`.

Name hiding

The scope of an entity typically encompasses more program text than the declaration space of the entity. In particular, the scope of an entity may include declarations that introduce new declaration spaces containing entities of the same name. Such declarations cause the original entity to become *hidden*. Conversely, an entity is said to be *visible* when it is not hidden.

Name hiding occurs when scopes overlap through nesting and when scopes overlap through inheritance. The characteristics of the two types of hiding are described in the following sections.

Hiding through nesting

Name hiding through nesting can occur as a result of nesting namespaces or types within namespaces, as a result of nesting types within classes or structs, and as a result of parameter and local variable declarations.

In the example

```
class A
{
    int i = 0;

    void F() {
        int i = 1;
    }

    void G() {
        i = 1;
    }
}
```

within the `F` method, the instance variable `i` is hidden by the local variable `i`, but within the `G` method, `i` still refers to the instance variable.

When a name in an inner scope hides a name in an outer scope, it hides all overloaded occurrences of that name. In the example

```

class Outer
{
    static void F(int i) {}

    static void F(string s) {}

    class Inner
    {
        void G() {
            F(1);           // Invokes Outer.Inner.F
            F("Hello");      // Error
        }

        static void F(long l) {}
    }
}

```

the call `F(1)` invokes the `F` declared in `Inner` because all outer occurrences of `F` are hidden by the inner declaration. For the same reason, the call `F("Hello")` results in a compile-time error.

Hiding through inheritance

Name hiding through inheritance occurs when classes or structs redeclare names that were inherited from base classes. This type of name hiding takes one of the following forms:

- A constant, field, property, event, or type introduced in a class or struct hides all base class members with the same name.
- A method introduced in a class or struct hides all non-method base class members with the same name, and all base class methods with the same signature (method name and parameter count, modifiers, and types).
- An indexer introduced in a class or struct hides all base class indexers with the same signature (parameter count and types).

The rules governing operator declarations ([Operators](#)) make it impossible for a derived class to declare an operator with the same signature as an operator in a base class. Thus, operators never hide one another.

Contrary to hiding a name from an outer scope, hiding an accessible name from an inherited scope causes a warning to be reported. In the example

```

class Base
{
    public void F() {}
}

class Derived: Base
{
    public void F() {}           // Warning, hiding an inherited name
}

```

the declaration of `F` in `Derived` causes a warning to be reported. Hiding an inherited name is specifically not an error, since that would preclude separate evolution of base classes. For example, the above situation might have come about because a later version of `Base` introduced an `F` method that wasn't present in an earlier version of the class. Had the above situation been an error, then any change made to a base class in a separately versioned class library could potentially cause derived classes to become invalid.

The warning caused by hiding an inherited name can be eliminated through use of the `new` modifier:

```

class Base
{
    public void F() {}
}

class Derived: Base
{
    new public void F() {}
}

```

The `new` modifier indicates that the `F` in `Derived` is "new", and that it is indeed intended to hide the inherited member.

A declaration of a new member hides an inherited member only within the scope of the new member.

```

class Base
{
    public static void F() {}
}

class Derived: Base
{
    new private static void F() {}    // Hides Base.F in Derived only
}

class MoreDerived: Derived
{
    static void G() { F(); }          // Invokes Base.F
}

```

In the example above, the declaration of `F` in `Derived` hides the `F` that was inherited from `Base`, but since the new `F` in `Derived` has private access, its scope does not extend to `MoreDerived`. Thus, the call `F()` in `MoreDerived.G` is valid and will invoke `Base.F`.

Namespace and type names

Several contexts in a C# program require a *namespace_name* or a *type_name* to be specified.

```

namespace_name
    : namespace_or_type_name
    ;

type_name
    : namespace_or_type_name
    ;

namespace_or_type_name
    : identifier type_argument_list?
    | namespace_or_type_name '.' identifier type_argument_list?
    | qualified_alias_member
    ;

```

A *namespace_name* is a *namespace_or_type_name* that refers to a namespace. Following resolution as described below, the *namespace_or_type_name* of a *namespace_name* must refer to a namespace, or otherwise a compile-time error occurs. No type arguments ([Type arguments](#)) can be present in a *namespace_name* (only types can have type arguments).

A *type_name* is a *namespace_or_type_name* that refers to a type. Following resolution as described below, the *namespace_or_type_name* of a *type_name* must refer to a type, or otherwise a compile-time error occurs.

If the *namespace_or_type_name* is a qualified-alias-member its meaning is as described in [Namespace alias qualifiers](#). Otherwise, a *namespace_or_type_name* has one of four forms:

- `I`
- `I<A1, ..., Ak>`
- `N.I`
- `N.I<A1, ..., Ak>`

where `I` is a single identifier, `N` is a *namespace_or_type_name* and `<A1, ..., Ak>` is an optional *type_argument_list*. When no *type_argument_list* is specified, consider `k` to be zero.

The meaning of a *namespace_or_type_name* is determined as follows:

- If the *namespace_or_type_name* is of the form `I` or of the form `I<A1, ..., Ak>`:
 - If `k` is zero and the *namespace_or_type_name* appears within a generic method declaration ([Methods](#)) and if that declaration includes a type parameter ([Type parameters](#)) with name `I`, then the *namespace_or_type_name* refers to that type parameter.
 - Otherwise, if the *namespace_or_type_name* appears within a type declaration, then for each instance type `T` ([The instance type](#)), starting with the instance type of that type declaration and continuing with the instance type of each enclosing class or struct declaration (if any):
 - If `k` is zero and the declaration of `T` includes a type parameter with name `I`, then the *namespace_or_type_name* refers to that type parameter.
 - Otherwise, if the *namespace_or_type_name* appears within the body of the type declaration, and `T` or any of its base types contain a nested accessible type having name `I` and `k` type parameters, then the *namespace_or_type_name* refers to that type constructed with the given type arguments. If there is more than one such type, the type declared within the more derived type is selected. Note that non-type members (constants, fields, methods, properties, indexers, operators, instance constructors, destructors, and static constructors) and type members with a different number of type parameters are ignored when determining the meaning of the *namespace_or_type_name*.
 - If the previous steps were unsuccessful then, for each namespace `N`, starting with the namespace in which the *namespace_or_type_name* occurs, continuing with each enclosing namespace (if any), and ending with the global namespace, the following steps are evaluated until an entity is located:
 - If `k` is zero and `I` is the name of a namespace in `N`, then:
 - If the location where the *namespace_or_type_name* occurs is enclosed by a namespace declaration for `N` and the namespace declaration contains an *extern_alias_directive* or *using_alias_directive* that associates the name `I` with a namespace or type, then the *namespace_or_type_name* is ambiguous and a compile-time error occurs.
 - Otherwise, the *namespace_or_type_name* refers to the namespace named `I` in `N`.
 - Otherwise, if `N` contains an accessible type having name `I` and `k` type parameters, then:
 - If `k` is zero and the location where the *namespace_or_type_name* occurs is enclosed by a namespace declaration for `N` and the namespace declaration contains an *extern_alias_directive* or *using_alias_directive* that associates the name `I` with a namespace or type, then the *namespace_or_type_name* is ambiguous and a compile-time error occurs.
 - Otherwise, the *namespace_or_type_name* refers to the type constructed with the given type arguments.
 - Otherwise, if the location where the *namespace_or_type_name* occurs is enclosed by a namespace declaration for `N`:
 - If `k` is zero and the namespace declaration contains an *extern_alias_directive* or *using_alias_directive* that associates the name `I` with an imported namespace or type,

then the *namespace_or_type_name* refers to that namespace or type.

- Otherwise, if the namespaces and type declarations imported by the *using_namespace_directives* and *using_alias_directives* of the namespace declaration contain exactly one accessible type having name `I` and `K` type parameters, then the *namespace_or_type_name* refers to that type constructed with the given type arguments.
- Otherwise, if the namespaces and type declarations imported by the *using_namespace_directives* and *using_alias_directives* of the namespace declaration contain more than one accessible type having name `I` and `K` type parameters, then the *namespace_or_type_name* is ambiguous and an error occurs.
- Otherwise, the *namespace_or_type_name* is undefined and a compile-time error occurs.
- Otherwise, the *namespace_or_type_name* is of the form `N.I` or of the form `N.I<A1, ..., Ak>`. `N` is first resolved as a *namespace_or_type_name*. If the resolution of `N` is not successful, a compile-time error occurs. Otherwise, `N.I` or `N.I<A1, ..., Ak>` is resolved as follows:
 - If `K` is zero and `N` refers to a namespace and `N` contains a nested namespace with name `I`, then the *namespace_or_type_name* refers to that nested namespace.
 - Otherwise, if `N` refers to a namespace and `N` contains an accessible type having name `I` and `K` type parameters, then the *namespace_or_type_name* refers to that type constructed with the given type arguments.
 - Otherwise, if `N` refers to a (possibly constructed) class or struct type and `N` or any of its base classes contain a nested accessible type having name `I` and `K` type parameters, then the *namespace_or_type_name* refers to that type constructed with the given type arguments. If there is more than one such type, the type declared within the more derived type is selected. Note that if the meaning of `N.I` is being determined as part of resolving the base class specification of `N` then the direct base class of `N` is considered to be object ([Base classes](#)).
 - Otherwise, `N.I` is an invalid *namespace_or_type_name*, and a compile-time error occurs.

A *namespace_or_type_name* is permitted to reference a static class ([Static classes](#)) only if

- The *namespace_or_type_name* is the `T` in a *namespace_or_type_name* of the form `T.I`, or
- The *namespace_or_type_name* is the `T` in a *typeof_expression* ([Argument lists1](#)) of the form `typeof(T)`.

Fully qualified names

Every namespace and type has a **fully qualified name**, which uniquely identifies the namespace or type amongst all others. The fully qualified name of a namespace or type `N` is determined as follows:

- If `N` is a member of the global namespace, its fully qualified name is `N`.
- Otherwise, its fully qualified name is `S.N`, where `S` is the fully qualified name of the namespace or type in which `N` is declared.

In other words, the fully qualified name of `N` is the complete hierarchical path of identifiers that lead to `N`, starting from the global namespace. Because every member of a namespace or type must have a unique name, it follows that the fully qualified name of a namespace or type is always unique.

The example below shows several namespace and type declarations along with their associated fully qualified names.

```

class A {}           // A

namespace X          // X
{
    class B          // X.B
    {
        class C {}   // X.B.C
    }

    namespace Y       // X.Y
    {
        class D {}   // X.Y.D
    }
}

namespace X.Y         // X.Y
{
    class E {}        // X.Y.E
}

```

Automatic memory management

C# employs automatic memory management, which frees developers from manually allocating and freeing the memory occupied by objects. Automatic memory management policies are implemented by a *garbage collector*. The memory management life cycle of an object is as follows:

1. When the object is created, memory is allocated for it, the constructor is run, and the object is considered live.
2. If the object, or any part of it, cannot be accessed by any possible continuation of execution, other than the running of destructors, the object is considered no longer in use, and it becomes eligible for destruction. The C# compiler and the garbage collector may choose to analyze code to determine which references to an object may be used in the future. For instance, if a local variable that is in scope is the only existing reference to an object, but that local variable is never referred to in any possible continuation of execution from the current execution point in the procedure, the garbage collector may (but is not required to) treat the object as no longer in use.
3. Once the object is eligible for destruction, at some unspecified later time the destructor ([Destructors](#)) (if any) for the object is run. Under normal circumstances the destructor for the object is run once only, though implementation-specific APIs may allow this behavior to be overridden.
4. Once the destructor for an object is run, if that object, or any part of it, cannot be accessed by any possible continuation of execution, including the running of destructors, the object is considered inaccessible and the object becomes eligible for collection.
5. Finally, at some time after the object becomes eligible for collection, the garbage collector frees the memory associated with that object.

The garbage collector maintains information about object usage, and uses this information to make memory management decisions, such as where in memory to locate a newly created object, when to relocate an object, and when an object is no longer in use or inaccessible.

Like other languages that assume the existence of a garbage collector, C# is designed so that the garbage collector may implement a wide range of memory management policies. For instance, C# does not require that destructors be run or that objects be collected as soon as they are eligible, or that destructors be run in any particular order, or on any particular thread.

The behavior of the garbage collector can be controlled, to some degree, via static methods on the class `System.GC`. This class can be used to request a collection to occur, destructors to be run (or not run), and so forth.

Since the garbage collector is allowed wide latitude in deciding when to collect objects and run destructors, a conforming implementation may produce output that differs from that shown by the following code. The program

```
using System;

class A
{
    ~A() {
        Console.WriteLine("Destruct instance of A");
    }
}

class B
{
    object Ref;

    public B(object o) {
        Ref = o;
    }

    ~B() {
        Console.WriteLine("Destruct instance of B");
    }
}

class Test
{
    static void Main() {
        B b = new B(new A());
        b = null;
        GC.Collect();
        GC.WaitForPendingFinalizers();
    }
}
```

creates an instance of class `A` and an instance of class `B`. These objects become eligible for garbage collection when the variable `b` is assigned the value `null`, since after this time it is impossible for any user-written code to access them. The output could be either

```
Destruct instance of A
Destruct instance of B
```

or

```
Destruct instance of B
Destruct instance of A
```

because the language imposes no constraints on the order in which objects are garbage collected.

In subtle cases, the distinction between "eligible for destruction" and "eligible for collection" can be important. For example,


```

using System;

class A
{
    ~A() {
        Console.WriteLine("Destruct instance of A");
    }

    public void F() {
        Console.WriteLine("A.F");
        Test.RefA = this;
    }
}

class B
{
    public A Ref;

    ~B() {
        Console.WriteLine("Destruct instance of B");
        Ref.F();
    }
}

class Test
{
    public static A RefA;
    public static B RefB;

    static void Main() {
        RefB = new B();
        RefA = new A();
        RefB.Ref = RefA;
        RefB = null;
        RefA = null;

        // A and B now eligible for destruction
        GC.Collect();
        GC.WaitForPendingFinalizers();

        // B now eligible for collection, but A is not
        if (RefA != null)
            Console.WriteLine("RefA is not null");
    }
}

```

In the above program, if the garbage collector chooses to run the destructor of `A` before the destructor of `B`, then the output of this program might be:

```

Destruct instance of A
Destruct instance of B
A.F
RefA is not null

```

Note that although the instance of `A` was not in use and `A`'s destructor was run, it is still possible for methods of `A` (in this case, `F`) to be called from another destructor. Also, note that running of a destructor may cause an object to become usable from the mainline program again. In this case, the running of `B`'s destructor caused an instance of `A` that was previously not in use to become accessible from the live reference `Test.RefA`. After the call to `WaitForPendingFinalizers`, the instance of `B` is eligible for collection, but the instance of `A` is not, because of the reference `Test.RefA`.

To avoid confusion and unexpected behavior, it is generally a good idea for destructors to only perform cleanup

on data stored in their object's own fields, and not to perform any actions on referenced objects or static fields.

An alternative to using destructors is to let a class implement the `System.IDisposable` interface. This allows the client of the object to determine when to release the resources of the object, typically by accessing the object as a resource in a `using` statement ([The using statement](#)).

Execution order

Execution of a C# program proceeds such that the side effects of each executing thread are preserved at critical execution points. A *side effect* is defined as a read or write of a volatile field, a write to a non-volatile variable, a write to an external resource, and the throwing of an exception. The critical execution points at which the order of these side effects must be preserved are references to volatile fields ([Volatile fields](#)), `lock` statements ([The lock statement](#)), and thread creation and termination. The execution environment is free to change the order of execution of a C# program, subject to the following constraints:

- Data dependence is preserved within a thread of execution. That is, the value of each variable is computed as if all statements in the thread were executed in original program order.
- Initialization ordering rules are preserved ([Field initialization](#) and [Variable initializers](#)).
- The ordering of side effects is preserved with respect to volatile reads and writes ([Volatile fields](#)). Additionally, the execution environment need not evaluate part of an expression if it can deduce that that expression's value is not used and that no needed side effects are produced (including any caused by calling a method or accessing a volatile field). When program execution is interrupted by an asynchronous event (such as an exception thrown by another thread), it is not guaranteed that the observable side effects are visible in the original program order.

Types

12/28/2021 • 33 minutes to read • [Edit Online](#)

The types of the C# language are divided into two main categories: *value types* and *reference types*. Both value types and reference types may be *generic types*, which take one or more *type parameters*. Type parameters can designate both value types and reference types.

```
type
    : value_type
    | reference_type
    | type_parameter
    | type_unsafe
    ;
```

The final category of types, pointers, is available only in unsafe code. This is discussed further in [Pointer types](#).

Value types differ from reference types in that variables of the value types directly contain their data, whereas variables of the reference types store *references* to their data, the latter being known as *objects*. With reference types, it is possible for two variables to reference the same object, and thus possible for operations on one variable to affect the object referenced by the other variable. With value types, the variables each have their own copy of the data, and it is not possible for operations on one to affect the other.

C#'s type system is unified such that a value of any type can be treated as an object. Every type in C# directly or indirectly derives from the `object` class type, and `object` is the ultimate base class of all types. Values of reference types are treated as objects simply by viewing the values as type `object`. Values of value types are treated as objects by performing boxing and unboxing operations ([Boxing and unboxing](#)).

Value types

A value type is either a struct type or an enumeration type. C# provides a set of predefined struct types called the *simple types*. The simple types are identified through reserved words.

```

value_type
    : struct_type
    | enum_type
    ;

struct_type
    : type_name
    | simple_type
    | nullable_type
    ;

simple_type
    : numeric_type
    | 'bool'
    ;

numeric_type
    : integral_type
    | floating_point_type
    | 'decimal'
    ;

integral_type
    : 'sbyte'
    | 'byte'
    | 'short'
    | 'ushort'
    | 'int'
    | 'uint'
    | 'long'
    | 'ulong'
    | 'char'
    ;

floating_point_type
    : 'float'
    | 'double'
    ;

nullable_type
    : non_nullable_value_type '?'
    ;

non_nullable_value_type
    : type
    ;

enum_type
    : type_name
    ;

```

Unlike a variable of a reference type, a variable of a value type can contain the value `null` only if the value type is a nullable type. For every non-nullable value type there is a corresponding nullable value type denoting the same set of values plus the value `null`.

Assignment to a variable of a value type creates a copy of the value being assigned. This differs from assignment to a variable of a reference type, which copies the reference but not the object identified by the reference.

The `System.ValueType` type

All value types implicitly inherit from the class `System.ValueType`, which, in turn, inherits from class `object`. It is not possible for any type to derive from a value type, and value types are thus implicitly sealed ([Sealed classes](#)).

Note that `System.ValueType` is not itself a *value_type*. Rather, it is a *class_type* from which all *value_types* are

automatically derived.

Default constructors

All value types implicitly declare a public parameterless instance constructor called the **default constructor**. The default constructor returns a zero-initialized instance known as the **default value** for the value type:

- For all *simple types*, the default value is the value produced by a bit pattern of all zeros:
 - For `sbyte`, `byte`, `short`, `ushort`, `int`, `uint`, `long`, and `ulong`, the default value is `0`.
 - For `char`, the default value is `'\x0000'`.
 - For `float`, the default value is `0.0f`.
 - For `double`, the default value is `0.0d`.
 - For `decimal`, the default value is `0.0m`.
 - For `bool`, the default value is `false`.
- For an *enum type* `E`, the default value is `0`, converted to the type `E`.
- For a *struct type*, the default value is the value produced by setting all value type fields to their default value and all reference type fields to `null`.
- For a *nullable type* the default value is an instance for which the `HasValue` property is false and the `Value` property is undefined. The default value is also known as the **null value** of the nullable type.

Like any other instance constructor, the default constructor of a value type is invoked using the `new` operator. For efficiency reasons, this requirement is not intended to actually have the implementation generate a constructor call. In the example below, variables `i` and `j` are both initialized to zero.

```
class A
{
    void F() {
        int i = 0;
        int j = new int();
    }
}
```

Because every value type implicitly has a public parameterless instance constructor, it is not possible for a struct type to contain an explicit declaration of a parameterless constructor. A struct type is however permitted to declare parameterized instance constructors ([Constructors](#)).

Struct types

A struct type is a value type that can declare constants, fields, methods, properties, indexers, operators, instance constructors, static constructors, and nested types. The declaration of struct types is described in [Struct declarations](#).

Simple types

C# provides a set of predefined struct types called the **simple types**. The simple types are identified through reserved words, but these reserved words are simply aliases for predefined struct types in the `System` namespace, as described in the table below.

RESERVED WORD	ALIASED TYPE
<code>sbyte</code>	<code>System.SByte</code>
<code>byte</code>	<code>System.Byte</code>
<code>short</code>	<code>System.Int16</code>

RESERVED WORD	ALIASED TYPE
<code>ushort</code>	<code>System.UInt16</code>
<code>int</code>	<code>System.Int32</code>
<code>uint</code>	<code>System.UInt32</code>
<code>long</code>	<code>System.Int64</code>
<code>ulong</code>	<code>System.UInt64</code>
<code>char</code>	<code>System.Char</code>
<code>float</code>	<code>System.Single</code>
<code>double</code>	<code>System.Double</code>
<code>bool</code>	<code>System.Boolean</code>
<code>decimal</code>	<code>System.Decimal</code>

Because a simple type aliases a struct type, every simple type has members. For example, `int` has the members declared in `System.Int32` and the members inherited from `System.Object`, and the following statements are permitted:

```
int i = int.MaxValue;           // System.Int32.MaxValue constant
string s = i.ToString();       // System.Int32.ToString() instance method
string t = 123.ToString();     // System.Int32.ToString() instance method
```

The simple types differ from other struct types in that they permit certain additional operations:

- Most simple types permit values to be created by writing *literals* ([Literals](#)). For example, `123` is a literal of type `int` and `'a'` is a literal of type `char`. C# makes no provision for literals of struct types in general, and non-default values of other struct types are ultimately always created through instance constructors of those struct types.
- When the operands of an expression are all simple type constants, it is possible for the compiler to evaluate the expression at compile-time. Such an expression is known as a *constant expression* ([Constant expressions](#)). Expressions involving operators defined by other struct types are not considered to be constant expressions.
- Through `const` declarations it is possible to declare constants of the simple types ([Constants](#)). It is not possible to have constants of other struct types, but a similar effect is provided by `static readonly` fields.
- Conversions involving simple types can participate in evaluation of conversion operators defined by other struct types, but a user-defined conversion operator can never participate in evaluation of another user-defined operator ([Evaluation of user-defined conversions](#)).

Integral types

C# supports nine integral types: `sbyte`, `byte`, `short`, `ushort`, `int`, `uint`, `long`, `ulong`, and `char`. The integral types have the following sizes and ranges of values:

- The `sbyte` type represents signed 8-bit integers with values between -128 and 127.
- The `byte` type represents unsigned 8-bit integers with values between 0 and 255.

- The `short` type represents signed 16-bit integers with values between -32768 and 32767.
- The `ushort` type represents unsigned 16-bit integers with values between 0 and 65535.
- The `int` type represents signed 32-bit integers with values between -2147483648 and 2147483647.
- The `uint` type represents unsigned 32-bit integers with values between 0 and 4294967295.
- The `long` type represents signed 64-bit integers with values between -9223372036854775808 and 9223372036854775807.
- The `ulong` type represents unsigned 64-bit integers with values between 0 and 18446744073709551615.
- The `char` type represents unsigned 16-bit integers with values between 0 and 65535. The set of possible values for the `char` type corresponds to the Unicode character set. Although `char` has the same representation as `ushort`, not all operations permitted on one type are permitted on the other.

The integral-type unary and binary operators always operate with signed 32-bit precision, unsigned 32-bit precision, signed 64-bit precision, or unsigned 64-bit precision:

- For the unary `+` and `~` operators, the operand is converted to type `T`, where `T` is the first of `int`, `uint`, `long`, and `ulong` that can fully represent all possible values of the operand. The operation is then performed using the precision of type `T`, and the type of the result is `T`.
- For the unary `-` operator, the operand is converted to type `T`, where `T` is the first of `int` and `long` that can fully represent all possible values of the operand. The operation is then performed using the precision of type `T`, and the type of the result is `T`. The unary `-` operator cannot be applied to operands of type `ulong`.
- For the binary `+`, `-`, `*`, `/`, `%`, `&`, `^`, `|`, `==`, `!=`, `>`, `<`, `>=`, and `<=` operators, the operands are converted to type `T`, where `T` is the first of `int`, `uint`, `long`, and `ulong` that can fully represent all possible values of both operands. The operation is then performed using the precision of type `T`, and the type of the result is `T` (or `bool` for the relational operators). It is not permitted for one operand to be of type `long` and the other to be of type `ulong` with the binary operators.
- For the binary `<<` and `>>` operators, the left operand is converted to type `T`, where `T` is the first of `int`, `uint`, `long`, and `ulong` that can fully represent all possible values of the operand. The operation is then performed using the precision of type `T`, and the type of the result is `T`.

The `char` type is classified as an integral type, but it differs from the other integral types in two ways:

- There are no implicit conversions from other types to the `char` type. In particular, even though the `sbyte`, `byte`, and `ushort` types have ranges of values that are fully representable using the `char` type, implicit conversions from `sbyte`, `byte`, or `ushort` to `char` do not exist.
- Constants of the `char` type must be written as *character literals* or as *integer literals* in combination with a cast to type `char`. For example, `(char)10` is the same as `'\x000A'`.

The `checked` and `unchecked` operators and statements are used to control overflow checking for integral-type arithmetic operations and conversions ([The checked and unchecked operators](#)). In a `checked` context, an overflow produces a compile-time error or causes a `System.OverflowException` to be thrown. In an `unchecked` context, overflows are ignored and any high-order bits that do not fit in the destination type are discarded.

Floating point types

C# supports two floating point types: `float` and `double`. The `float` and `double` types are represented using the 32-bit single-precision and 64-bit double-precision IEEE 754 formats, which provide the following sets of values:

- Positive zero and negative zero. In most situations, positive zero and negative zero behave identically as the simple value zero, but certain operations distinguish between the two ([Division operator](#)).
- Positive infinity and negative infinity. Infinities are produced by such operations as dividing a non-zero number by zero. For example, `1.0 / 0.0` yields positive infinity, and `-1.0 / 0.0` yields negative infinity.

- The **Not-a-Number** value, often abbreviated NaN. NaNs are produced by invalid floating-point operations, such as dividing zero by zero.
- The finite set of non-zero values of the form $s * m * 2^e$, where s is 1 or -1, and m and e are determined by the particular floating-point type: For `float`, $0 < m < 2^{24}$ and $-149 \leq e \leq 104$, and for `double`, $0 < m < 2^{53}$ and $-1075 \leq e \leq 970$. Denormalized floating-point numbers are considered valid non-zero values.

The `float` type can represent values ranging from approximately $1.5 * 10^{-45}$ to $3.4 * 10^{38}$ with a precision of 7 digits.

The `double` type can represent values ranging from approximately $5.0 * 10^{-324}$ to $1.7 * 10^{308}$ with a precision of 15-16 digits.

If one of the operands of a binary operator is of a floating-point type, then the other operand must be of an integral type or a floating-point type, and the operation is evaluated as follows:

- If one of the operands is of an integral type, then that operand is converted to the floating-point type of the other operand.
- Then, if either of the operands is of type `double`, the other operand is converted to `double`, the operation is performed using at least `double` range and precision, and the type of the result is `double` (or `bool` for the relational operators).
- Otherwise, the operation is performed using at least `float` range and precision, and the type of the result is `float` (or `bool` for the relational operators).

The floating-point operators, including the assignment operators, never produce exceptions. Instead, in exceptional situations, floating-point operations produce zero, infinity, or NaN, as described below:

- If the result of a floating-point operation is too small for the destination format, the result of the operation becomes positive zero or negative zero.
- If the result of a floating-point operation is too large for the destination format, the result of the operation becomes positive infinity or negative infinity.
- If a floating-point operation is invalid, the result of the operation becomes NaN.
- If one or both operands of a floating-point operation is NaN, the result of the operation becomes NaN.

Floating-point operations may be performed with higher precision than the result type of the operation. For example, some hardware architectures support an "extended" or "long double" floating-point type with greater range and precision than the `double` type, and implicitly perform all floating-point operations using this higher precision type. Only at excessive cost in performance can such hardware architectures be made to perform floating-point operations with less precision, and rather than require an implementation to forfeit both performance and precision, C# allows a higher precision type to be used for all floating-point operations. Other than delivering more precise results, this rarely has any measurable effects. However, in expressions of the form $x * y / z$, where the multiplication produces a result that is outside the `double` range, but the subsequent division brings the temporary result back into the `double` range, the fact that the expression is evaluated in a higher range format may cause a finite result to be produced instead of an infinity.

The decimal type

The `decimal` type is a 128-bit data type suitable for financial and monetary calculations. The `decimal` type can represent values ranging from $1.0 * 10^{-28}$ to approximately $7.9 * 10^{28}$ with 28-29 significant digits.

The finite set of values of type `decimal` are of the form $(-1)^s * c * 10^{-e}$, where the sign s is 0 or 1, the coefficient c is given by $0 \leq c < 2^{96}$, and the scale e is such that $0 \leq e \leq 28$. The `decimal` type does not support signed zeros, infinities, or NaN's. A `decimal` is represented as a 96-bit integer scaled by a power of ten. For `decimal`s with an absolute value less than $1.0m$, the value is exact to the 28th decimal place, but no further. For `decimal`s with an absolute value greater than or equal to $1.0m$, the value is exact to 28 or 29 digits.

Contrary to the `float` and `double` data types, decimal fractional numbers such as 0.1 can be represented exactly in the `decimal` representation. In the `float` and `double` representations, such numbers are often infinite fractions, making those representations more prone to round-off errors.

If one of the operands of a binary operator is of type `decimal`, then the other operand must be of an integral type or of type `decimal`. If an integral type operand is present, it is converted to `decimal` before the operation is performed.

The result of an operation on values of type `decimal` is that which would result from calculating an exact result (preserving scale, as defined for each operator) and then rounding to fit the representation. Results are rounded to the nearest representable value, and, when a result is equally close to two representable values, to the value that has an even number in the least significant digit position (this is known as "banker's rounding"). A zero result always has a sign of 0 and a scale of 0.

If a decimal arithmetic operation produces a value less than or equal to $5 * 10^{-29}$ in absolute value, the result of the operation becomes zero. If a `decimal` arithmetic operation produces a result that is too large for the `decimal` format, a `System.OverflowException` is thrown.

The `decimal` type has greater precision but smaller range than the floating-point types. Thus, conversions from the floating-point types to `decimal` might produce overflow exceptions, and conversions from `decimal` to the floating-point types might cause loss of precision. For these reasons, no implicit conversions exist between the floating-point types and `decimal`, and without explicit casts, it is not possible to mix floating-point and `decimal` operands in the same expression.

The bool type

The `bool` type represents boolean logical quantities. The possible values of type `bool` are `true` and `false`.

No standard conversions exist between `bool` and other types. In particular, the `bool` type is distinct and separate from the integral types, and a `bool` value cannot be used in place of an integral value, and vice versa.

In the C and C++ languages, a zero integral or floating-point value, or a null pointer can be converted to the boolean value `false`, and a non-zero integral or floating-point value, or a non-null pointer can be converted to the boolean value `true`. In C#, such conversions are accomplished by explicitly comparing an integral or floating-point value to zero, or by explicitly comparing an object reference to `null`.

Enumeration types

An enumeration type is a distinct type with named constants. Every enumeration type has an underlying type, which must be `byte`, `sbyte`, `short`, `ushort`, `int`, `uint`, `long` or `ulong`. The set of values of the enumeration type is the same as the set of values of the underlying type. Values of the enumeration type are not restricted to the values of the named constants. Enumeration types are defined through enumeration declarations ([Enum declarations](#)).

Nullable types

A nullable type can represent all values of its *underlying type* plus an additional null value. A nullable type is written `T?`, where `T` is the underlying type. This syntax is shorthand for `System.Nullable<T>`, and the two forms can be used interchangeably.

A *non-nullable value type* conversely is any value type other than `System.Nullable<T>` and its shorthand `T?` (for any `T`), plus any type parameter that is constrained to be a non-nullable value type (that is, any type parameter with a `struct` constraint). The `System.Nullable<T>` type specifies the value type constraint for `T` ([Type parameter constraints](#)), which means that the underlying type of a nullable type can be any non-nullable value type. The underlying type of a nullable type cannot be a nullable type or a reference type. For example, `int??` and `string?` are invalid types.

An instance of a nullable type `T?` has two public read-only properties:

- A `HasValue` property of type `bool`
- A `Value` property of type `T`

An instance for which `HasValue` is true is said to be non-null. A non-null instance contains a known value and `Value` returns that value.

An instance for which `HasValue` is false is said to be null. A null instance has an undefined value. Attempting to read the `Value` of a null instance causes a `System.InvalidOperationException` to be thrown. The process of accessing the `Value` property of a nullable instance is referred to as *unwrapping*.

In addition to the default constructor, every nullable type `T?` has a public constructor that takes a single argument of type `T`. Given a value `x` of type `T`, a constructor invocation of the form

```
new T?(x)
```

creates a non-null instance of `T?` for which the `Value` property is `x`. The process of creating a non-null instance of a nullable type for a given value is referred to as *wrapping*.

Implicit conversions are available from the `null` literal to `T?` ([Null literal conversions](#)) and from `T` to `T?` ([Implicit nullable conversions](#)).

Reference types

A reference type is a class type, an interface type, an array type, or a delegate type.

```

reference_type
    : class_type
    | interface_type
    | array_type
    | delegate_type
    ;

class_type
    : type_name
    | 'object'
    | 'dynamic'
    | 'string'
    ;

interface_type
    : type_name
    ;

array_type
    : non_array_type rank_specifier+
    ;

non_array_type
    : type
    ;

rank_specifier
    : '[' dim_separator* ']'
    ;

dim_separator
    : ','
    ;

delegate_type
    : type_name
    ;

```

A reference type value is a reference to an *instance* of the type, the latter known as an *object*. The special value `null` is compatible with all reference types and indicates the absence of an instance.

Class types

A class type defines a data structure that contains data members (constants and fields), function members (methods, properties, events, indexers, operators, instance constructors, destructors and static constructors), and nested types. Class types support inheritance, a mechanism whereby derived classes can extend and specialize base classes. Instances of class types are created using *object_creation_expressions* ([Object creation expressions](#)).

Class types are described in [Classes](#).

Certain predefined class types have special meaning in the C# language, as described in the table below.

CLASS TYPE	DESCRIPTION
<code>System.Object</code>	The ultimate base class of all other types. See The object type .
<code>System.String</code>	The string type of the C# language. See The string type .
<code>System.ValueType</code>	The base class of all value types. See The System.ValueType type .

CLASS TYPE	DESCRIPTION
<code>System.Enum</code>	The base class of all enum types. See Enums .
<code>System.Array</code>	The base class of all array types. See Arrays .
<code>System.Delegate</code>	The base class of all delegate types. See Delegates .
<code>System.Exception</code>	The base class of all exception types. See Exceptions .

The object type

The `object` class type is the ultimate base class of all other types. Every type in C# directly or indirectly derives from the `object` class type.

The keyword `object` is simply an alias for the predefined class `System.Object`.

The dynamic type

The `dynamic` type, like `object`, can reference any object. When operators are applied to expressions of type `dynamic`, their resolution is deferred until the program is run. Thus, if the operator cannot legally be applied to the referenced object, no error is given during compilation. Instead an exception will be thrown when resolution of the operator fails at run-time.

Its purpose is to allow dynamic binding, which is described in detail in [Dynamic binding](#).

`dynamic` is considered identical to `object` except in the following respects:

- Operations on expressions of type `dynamic` can be dynamically bound ([Dynamic binding](#)).
- Type inference ([Type inference](#)) will prefer `dynamic` over `object` if both are candidates.

Because of this equivalence, the following holds:

- There is an implicit identity conversion between `object` and `dynamic`, and between constructed types that are the same when replacing `dynamic` with `object`.
- Implicit and explicit conversions to and from `object` also apply to and from `dynamic`.
- Method signatures that are the same when replacing `dynamic` with `object` are considered the same signature.
- The type `dynamic` is indistinguishable from `object` at run-time.
- An expression of the type `dynamic` is referred to as a *dynamic expression*.

The string type

The `string` type is a sealed class type that inherits directly from `object`. Instances of the `string` class represent Unicode character strings.

Values of the `string` type can be written as string literals ([String literals](#)).

The keyword `string` is simply an alias for the predefined class `System.String`.

Interface types

An interface defines a contract. A class or struct that implements an interface must adhere to its contract. An interface may inherit from multiple base interfaces, and a class or struct may implement multiple interfaces.

Interface types are described in [Interfaces](#).

Array types

An array is a data structure that contains zero or more variables which are accessed through computed indices.

The variables contained in an array, also called the elements of the array, are all of the same type, and this type is called the element type of the array.

Array types are described in [Arrays](#).

Delegate types

A delegate is a data structure that refers to one or more methods. For instance methods, it also refers to their corresponding object instances.

The closest equivalent of a delegate in C or C++ is a function pointer, but whereas a function pointer can only reference static functions, a delegate can reference both static and instance methods. In the latter case, the delegate stores not only a reference to the method's entry point, but also a reference to the object instance on which to invoke the method.

Delegate types are described in [Delegates](#).

Boxing and unboxing

The concept of boxing and unboxing is central to C#'s type system. It provides a bridge between *value_types* and *reference_types* by permitting any value of a *value_type* to be converted to and from type `object`. Boxing and unboxing enables a unified view of the type system wherein a value of any type can ultimately be treated as an object.

Boxing conversions

A boxing conversion permits a *value_type* to be implicitly converted to a *reference_type*. The following boxing conversions exist:

- From any *value_type* to the type `object`.
- From any *value_type* to the type `System.ValueType`.
- From any *non_nullable_value_type* to any *interface_type* implemented by the *value_type*.
- From any *nullable_type* to any *interface_type* implemented by the underlying type of the *nullable_type*.
- From any *enum_type* to the type `System.Enum`.
- From any *nullable_type* with an underlying *enum_type* to the type `System.Enum`.
- Note that an implicit conversion from a type parameter will be executed as a boxing conversion if at run-time it ends up converting from a value type to a reference type ([Implicit conversions involving type parameters](#)).

Boxing a value of a *non_nullable_value_type* consists of allocating an object instance and copying the *non_nullable_value_type* value into that instance.

Boxing a value of a *nullable_type* produces a null reference if it is the `null` value (`HasValue` is `false`), or the result of unwrapping and boxing the underlying value otherwise.

The actual process of boxing a value of a *non_nullable_value_type* is best explained by imagining the existence of a generic **boxing class**, which behaves as if it were declared as follows:

```
sealed class Box<T>: System.ValueType
{
    T value;

    public Box(T t) {
        value = t;
    }
}
```

Boxing of a value `v` of type `T` now consists of executing the expression `new Box<T>(v)`, and returning the resulting instance as a value of type `object`. Thus, the statements

```
int i = 123;
object box = i;
```

conceptually correspond to

```
int i = 123;
object box = new Box<int>(i);
```

A boxing class like `Box<T>` above doesn't actually exist and the dynamic type of a boxed value isn't actually a class type. Instead, a boxed value of type `T` has the dynamic type `T`, and a dynamic type check using the `is` operator can simply reference type `T`. For example,

```
int i = 123;
object box = i;
if (box is int) {
    Console.WriteLine("Box contains an int");
}
```

will output the string "`Box contains an int`" on the console.

A boxing conversion implies making a copy of the value being boxed. This is different from a conversion of a *reference_type* to type `object`, in which the value continues to reference the same instance and simply is regarded as the less derived type `object`. For example, given the declaration

```
struct Point
{
    public int x, y;

    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }
}
```

the following statements

```
Point p = new Point(10, 10);
object box = p;
p.x = 20;
Console.WriteLine(((Point)box).x);
```

will output the value 10 on the console because the implicit boxing operation that occurs in the assignment of `p` to `box` causes the value of `p` to be copied. Had `Point` been declared a `class` instead, the value 20 would be output because `p` and `box` would reference the same instance.

Unboxing conversions

An unboxing conversion permits a *reference_type* to be explicitly converted to a *value_type*. The following unboxing conversions exist:

- From the type `object` to any *value_type*.
- From the type `System.ValueType` to any *value_type*.
- From any *interface_type* to any *non_nullable_value_type* that implements the *interface_type*.
- From any *interface_type* to any *nullable_type* whose underlying type implements the *interface_type*.
- From the type `System.Enum` to any *enum_type*.

- From the type `System.Enum` to any *nullable_type* with an underlying *enum_type*.
- Note that an explicit conversion to a type parameter will be executed as an unboxing conversion if at run-time it ends up converting from a reference type to a value type ([Explicit dynamic conversions](#)).

An unboxing operation to a *non_nullable_value_type* consists of first checking that the object instance is a boxed value of the given *non_nullable_value_type*, and then copying the value out of the instance.

Unboxing to a *nullable_type* produces the null value of the *nullable_type* if the source operand is `null`, or the wrapped result of unboxing the object instance to the underlying type of the *nullable_type* otherwise.

Referring to the imaginary boxing class described in the previous section, an unboxing conversion of an object `box` to a *value_type* `T` consists of executing the expression `((Box<T>)box).value`. Thus, the statements

```
object box = 123;
int i = (int)box;
```

conceptually correspond to

```
object box = new Box<int>(123);
int i = ((Box<int>)box).value;
```

For an unboxing conversion to a given *non_nullable_value_type* to succeed at run-time, the value of the source operand must be a reference to a boxed value of that *non_nullable_value_type*. If the source operand is `null`, a `System.NullReferenceException` is thrown. If the source operand is a reference to an incompatible object, a `System.InvalidCastException` is thrown.

For an unboxing conversion to a given *nullable_type* to succeed at run-time, the value of the source operand must be either `null` or a reference to a boxed value of the underlying *non_nullable_value_type* of the *nullable_type*. If the source operand is a reference to an incompatible object, a `System.InvalidCastException` is thrown.

Constructed types

A generic type declaration, by itself, denotes an *unbound generic type* that is used as a "blueprint" to form many different types, by way of applying *type arguments*. The type arguments are written within angle brackets (`<` and `>`) immediately following the name of the generic type. A type that includes at least one type argument is called a *constructed type*. A constructed type can be used in most places in the language in which a type name can appear. An unbound generic type can only be used within a *typeof_expression* ([The typeof operator](#)).

Constructed types can also be used in expressions as simple names ([Simple names](#)) or when accessing a member ([Member access](#)).

When a *namespace_or_type_name* is evaluated, only generic types with the correct number of type parameters are considered. Thus, it is possible to use the same identifier to identify different types, as long as the types have different numbers of type parameters. This is useful when mixing generic and non-generic classes in the same program:

```

namespace Widgets
{
    class Queue {...}
    class Queue<TElement> {...}
}

namespace MyApplication
{
    using Widgets;

    class X
    {
        Queue q1;           // Non-generic Widgets.Queue
        Queue<int> q2;       // Generic Widgets.Queue
    }
}

```

A *type_name* might identify a constructed type even though it doesn't specify type parameters directly. This can occur where a type is nested within a generic class declaration, and the instance type of the containing declaration is implicitly used for name lookup ([Nested types in generic classes](#)):

```

class Outer<T>
{
    public class Inner {...}

    public Inner i;           // Type of i is Outer<T>.Inner
}

```

In unsafe code, a constructed type cannot be used as an *unmanaged_type* ([Pointer types](#)).

Type arguments

Each argument in a type argument list is simply a *type*.

```

type_argument_list
    : '<' type_arguments '>'
    ;

type_arguments
    : type_argument (',' type_argument)*
    ;

type_argument
    : type
    ;

```

In unsafe code ([Unsafe code](#)), a *type_argument* may not be a pointer type. Each type argument must satisfy any constraints on the corresponding type parameter ([Type parameter constraints](#)).

Open and closed types

All types can be classified as either *open types* or *closed types*. An open type is a type that involves type parameters. More specifically:

- A type parameter defines an open type.
- An array type is an open type if and only if its element type is an open type.
- A constructed type is an open type if and only if one or more of its type arguments is an open type. A constructed nested type is an open type if and only if one or more of its type arguments or the type arguments of its containing type(s) is an open type.

A closed type is a type that is not an open type.

At run-time, all of the code within a generic type declaration is executed in the context of a closed constructed type that was created by applying type arguments to the generic declaration. Each type parameter within the generic type is bound to a particular run-time type. The run-time processing of all statements and expressions always occurs with closed types, and open types occur only during compile-time processing.

Each closed constructed type has its own set of static variables, which are not shared with any other closed constructed types. Since an open type does not exist at run-time, there are no static variables associated with an open type. Two closed constructed types are the same type if they are constructed from the same unbound generic type, and their corresponding type arguments are the same type.

Bound and unbound types

The term **unbound type** refers to a non-generic type or an unbound generic type. The term **bound type** refers to a non-generic type or a constructed type.

An unbound type refers to the entity declared by a type declaration. An unbound generic type is not itself a type, and cannot be used as the type of a variable, argument or return value, or as a base type. The only construct in which an unbound generic type can be referenced is the `typeof` expression ([The typeof operator](#)).

Satisfying constraints

Whenever a constructed type or generic method is referenced, the supplied type arguments are checked against the type parameter constraints declared on the generic type or method ([Type parameter constraints](#)). For each `where` clause, the type argument `A` that corresponds to the named type parameter is checked against each constraint as follows:

- If the constraint is a class type, an interface type, or a type parameter, let `C` represent that constraint with the supplied type arguments substituted for any type parameters that appear in the constraint. To satisfy the constraint, it must be the case that type `A` is convertible to type `C` by one of the following:
 - An identity conversion ([Identity conversion](#))
 - An implicit reference conversion ([Implicit reference conversions](#))
 - A boxing conversion ([Boxing conversions](#)), provided that type `A` is a non-nullable value type.
 - An implicit reference, boxing or type parameter conversion from a type parameter `A` to `C`.
- If the constraint is the reference type constraint (`class`), the type `A` must satisfy one of the following:
 - `A` is an interface type, class type, delegate type or array type. Note that `System.ValueType` and `System.Enum` are reference types that satisfy this constraint.
 - `A` is a type parameter that is known to be a reference type ([Type parameter constraints](#)).
- If the constraint is the value type constraint (`struct`), the type `A` must satisfy one of the following:
 - `A` is a struct type or enum type, but not a nullable type. Note that `System.ValueType` and `System.Enum` are reference types that do not satisfy this constraint.
 - `A` is a type parameter having the value type constraint ([Type parameter constraints](#)).
- If the constraint is the constructor constraint (`new()`), the type `A` must not be `abstract` and must have a public parameterless constructor. This is satisfied if one of the following is true:
 - `A` is a value type, since all value types have a public default constructor ([Default constructors](#)).
 - `A` is a type parameter having the constructor constraint ([Type parameter constraints](#)).
 - `A` is a type parameter having the value type constraint ([Type parameter constraints](#)).
 - `A` is a class that is not `abstract` and contains an explicitly declared `public` constructor with no parameters.
 - `A` is not `abstract` and has a default constructor ([Default constructors](#)).

A compile-time error occurs if one or more of a type parameter's constraints are not satisfied by the given type arguments.

Since type parameters are not inherited, constraints are never inherited either. In the example below, `D` needs to specify the constraint on its type parameter `T` so that `T` satisfies the constraint imposed by the base class `B<T>`. In contrast, class `E` need not specify a constraint, because `List<T>` implements `IEnumerable` for any `T`.

```
class B<T> where T: IEnumerable {...}

class D<T>: B<T> where T: IEnumerable {...}

class E<T>: B<List<T>> {...}
```

Type parameters

A type parameter is an identifier designating a value type or reference type that the parameter is bound to at run-time.

```
type_parameter
: identifier
;
```

Since a type parameter can be instantiated with many different actual type arguments, type parameters have slightly different operations and restrictions than other types. These include:

- A type parameter cannot be used directly to declare a base class ([Base class](#)) or interface ([Variant type parameter lists](#)).
- The rules for member lookup on type parameters depend on the constraints, if any, applied to the type parameter. They are detailed in [Member lookup](#).
- The available conversions for a type parameter depend on the constraints, if any, applied to the type parameter. They are detailed in [Implicit conversions involving type parameters](#) and [Explicit dynamic conversions](#).
- The literal `null` cannot be converted to a type given by a type parameter, except if the type parameter is known to be a reference type ([Implicit conversions involving type parameters](#)). However, a `default` expression ([Default value expressions](#)) can be used instead. In addition, a value with a type given by a type parameter can be compared with `null` using `==` and `!=` ([Reference type equality operators](#)) unless the type parameter has the value type constraint.
- A `new` expression ([Object creation expressions](#)) can only be used with a type parameter if the type parameter is constrained by a *constructor_constraint* or the value type constraint ([Type parameter constraints](#)).
- A type parameter cannot be used anywhere within an attribute.
- A type parameter cannot be used in a member access ([Member access](#)) or type name ([Namespace and type names](#)) to identify a static member or a nested type.
- In unsafe code, a type parameter cannot be used as an *unmanaged_type* ([Pointer types](#)).

As a type, type parameters are purely a compile-time construct. At run-time, each type parameter is bound to a run-time type that was specified by supplying a type argument to the generic type declaration. Thus, the type of a variable declared with a type parameter will, at run-time, be a closed constructed type ([Open and closed types](#)). The run-time execution of all statements and expressions involving type parameters uses the actual type that was supplied as the type argument for that parameter.

Expression tree types

Expression trees permit lambda expressions to be represented as data structures instead of executable code.

Expression trees are values of *expression tree types* of the form `System.Linq.Expressions.Expression<D>`,

where `D` is any delegate type. For the remainder of this specification we will refer to these types using the shorthand `Expression<D>`.

If a conversion exists from a lambda expression to a delegate type `D`, a conversion also exists to the expression tree type `Expression<D>`. Whereas the conversion of a lambda expression to a delegate type generates a delegate that references executable code for the lambda expression, conversion to an expression tree type creates an expression tree representation of the lambda expression.

Expression trees are efficient in-memory data representations of lambda expressions and make the structure of the lambda expression transparent and explicit.

Just like a delegate type `D`, `Expression<D>` is said to have parameter and return types, which are the same as those of `D`.

The following example represents a lambda expression both as executable code and as an expression tree.

Because a conversion exists to `Func<int,int>`, a conversion also exists to `Expression<Func<int,int>>`:

```
Func<int,int> del = x => x + 1;           // Code

Expression<Func<int,int>> exp = x => x + 1; // Data
```

Following these assignments, the delegate `del` references a method that returns `x + 1`, and the expression tree `exp` references a data structure that describes the expression `x => x + 1`.

The exact definition of the generic type `Expression<D>` as well as the precise rules for constructing an expression tree when a lambda expression is converted to an expression tree type, are both outside the scope of this specification.

Two things are important to make explicit:

- Not all lambda expressions can be converted to expression trees. For instance, lambda expressions with statement bodies, and lambda expressions containing assignment expressions cannot be represented. In these cases, a conversion still exists, but will fail at compile-time. These exceptions are detailed in [Anonymous function conversions](#).
- `Expression<D>` offers an instance method `Compile` which produces a delegate of type `D`:

```
Func<int,int> del2 = exp.Compile();
```

Invoking this delegate causes the code represented by the expression tree to be executed. Thus, given the definitions above, `del` and `del2` are equivalent, and the following two statements will have the same effect:

```
int i1 = del(1);

int i2 = del2(1);
```

After executing this code, `i1` and `i2` will both have the value `2`.

Variables

12/28/2021 • 32 minutes to read • [Edit Online](#)

Variables represent storage locations. Every variable has a type that determines what values can be stored in the variable. C# is a type-safe language, and the C# compiler guarantees that values stored in variables are always of the appropriate type. The value of a variable can be changed through assignment or through use of the `++` and `--` operators.

A variable must be *definitely assigned* ([Definite assignment](#)) before its value can be obtained.

As described in the following sections, variables are either *initially assigned* or *initially unassigned*. An initially assigned variable has a well-defined initial value and is always considered definitely assigned. An initially unassigned variable has no initial value. For an initially unassigned variable to be considered definitely assigned at a certain location, an assignment to the variable must occur in every possible execution path leading to that location.

Variable categories

C# defines seven categories of variables: static variables, instance variables, array elements, value parameters, reference parameters, output parameters, and local variables. The sections that follow describe each of these categories.

In the example

```
class A
{
    public static int x;
    int y;

    void F(int[] v, int a, ref int b, out int c) {
        int i = 1;
        c = a + b++;
    }
}
```

`x` is a static variable, `y` is an instance variable, `v[0]` is an array element, `a` is a value parameter, `b` is a reference parameter, `c` is an output parameter, and `i` is a local variable.

Static variables

A field declared with the `static` modifier is called a *static variable*. A static variable comes into existence before execution of the static constructor ([Static constructors](#)) for its containing type, and ceases to exist when the associated application domain ceases to exist.

The initial value of a static variable is the default value ([Default values](#)) of the variable's type.

For purposes of definite assignment checking, a static variable is considered initially assigned.

Instance variables

A field declared without the `static` modifier is called an *instance variable*.

Instance variables in classes

An instance variable of a class comes into existence when a new instance of that class is created, and ceases to exist when there are no references to that instance and the instance's destructor (if any) has executed.

The initial value of an instance variable of a class is the default value ([Default values](#)) of the variable's type.

For the purpose of definite assignment checking, an instance variable of a class is considered initially assigned.

Instance variables in structs

An instance variable of a struct has exactly the same lifetime as the struct variable to which it belongs. In other words, when a variable of a struct type comes into existence or ceases to exist, so too do the instance variables of the struct.

The initial assignment state of an instance variable of a struct is the same as that of the containing struct variable. In other words, when a struct variable is considered initially assigned, so too are its instance variables, and when a struct variable is considered initially unassigned, its instance variables are likewise unassigned.

Array elements

The elements of an array come into existence when an array instance is created, and cease to exist when there are no references to that array instance.

The initial value of each of the elements of an array is the default value ([Default values](#)) of the type of the array elements.

For the purpose of definite assignment checking, an array element is considered initially assigned.

Value parameters

A parameter declared without a `ref` or `out` modifier is a *value parameter*.

A value parameter comes into existence upon invocation of the function member (method, instance constructor, accessor, or operator) or anonymous function to which the parameter belongs, and is initialized with the value of the argument given in the invocation. A value parameter normally ceases to exist upon return of the function member or anonymous function. However, if the value parameter is captured by an anonymous function ([Anonymous function expressions](#)), its life time extends at least until the delegate or expression tree created from that anonymous function is eligible for garbage collection.

For the purpose of definite assignment checking, a value parameter is considered initially assigned.

Reference parameters

A parameter declared with a `ref` modifier is a *reference parameter*.

A reference parameter does not create a new storage location. Instead, a reference parameter represents the same storage location as the variable given as the argument in the function member or anonymous function invocation. Thus, the value of a reference parameter is always the same as the underlying variable.

The following definite assignment rules apply to reference parameters. Note the different rules for output parameters described in [Output parameters](#).

- A variable must be definitely assigned ([Definite assignment](#)) before it can be passed as a reference parameter in a function member or delegate invocation.
- Within a function member or anonymous function, a reference parameter is considered initially assigned.

Within an instance method or instance accessor of a struct type, the `this` keyword behaves exactly as a reference parameter of the struct type ([This access](#)).

Output parameters

A parameter declared with an `out` modifier is an *output parameter*.

An output parameter does not create a new storage location. Instead, an output parameter represents the same storage location as the variable given as the argument in the function member or delegate invocation. Thus, the value of an output parameter is always the same as the underlying variable.

The following definite assignment rules apply to output parameters. Note the different rules for reference parameters described in [Reference parameters](#).

- A variable need not be definitely assigned before it can be passed as an output parameter in a function member or delegate invocation.
- Following the normal completion of a function member or delegate invocation, each variable that was passed as an output parameter is considered assigned in that execution path.
- Within a function member or anonymous function, an output parameter is considered initially unassigned.
- Every output parameter of a function member or anonymous function must be definitely assigned ([Definite assignment](#)) before the function member or anonymous function returns normally.

Within an instance constructor of a struct type, the `this` keyword behaves exactly as an output parameter of the struct type ([This access](#)).

Local variables

A **local variable** is declared by a *local_variable_declaration*, which may occur in a *block*, a *for_statement*, a *switch_statement* or a *using_statement*, or by a *foreach_statement* or a *specific_catch_clause* for a *try_statement*.

The lifetime of a local variable is the portion of program execution during which storage is guaranteed to be reserved for it. This lifetime extends at least from entry into the *block*, *for_statement*, *switch_statement*, *using_statement*, *foreach_statement*, or *specific_catch_clause* with which it is associated, until execution of that *block*, *for_statement*, *switch_statement*, *using_statement*, *foreach_statement*, or *specific_catch_clause* ends in any way. (Entering an enclosed *block* or calling a method suspends, but does not end, execution of the current *block*, *for_statement*, *switch_statement*, *using_statement*, *foreach_statement*, or *specific_catch_clause*.) If the local variable is captured by an anonymous function ([Captured outer variables](#)), its lifetime extends at least until the delegate or expression tree created from the anonymous function, along with any other objects that come to reference the captured variable, are eligible for garbage collection.

If the parent *block*, *for_statement*, *switch_statement*, *using_statement*, *foreach_statement*, or *specific_catch_clause* is entered recursively, a new instance of the local variable is created each time, and its *local_variable_initializer*, if any, is evaluated each time.

A local variable introduced by a *local_variable_declaration* is not automatically initialized and thus has no default value. For the purpose of definite assignment checking, a local variable introduced by a *local_variable_declaration* is considered initially unassigned. A *local_variable_declaration* may include a *local_variable_initializer*, in which case the variable is considered definitely assigned only after the initializing expression ([Declaration statements](#)).

Within the scope of a local variable introduced by a *local_variable_declaration*, it is a compile-time error to refer to that local variable in a textual position that precedes its *local_variable_declarator*. If the local variable declaration is implicit ([Local variable declarations](#)), it is also an error to refer to the variable within its *local_variable_declarator*.

A local variable introduced by a *foreach_statement* or a *specific_catch_clause* is considered definitely assigned in its entire scope.

The actual lifetime of a local variable is implementation-dependent. For example, a compiler might statically determine that a local variable in a block is only used for a small portion of that block. Using this analysis, the compiler could generate code that results in the variable's storage having a shorter lifetime than its containing block.

The storage referred to by a local reference variable is reclaimed independently of the lifetime of that local reference variable ([Automatic memory management](#)).

Default values

The following categories of variables are automatically initialized to their default values:

- Static variables.
- Instance variables of class instances.
- Array elements.

The default value of a variable depends on the type of the variable and is determined as follows:

- For a variable of a *value_type*, the default value is the same as the value computed by the *value_type*'s default constructor ([Default constructors](#)).
- For a variable of a *reference_type*, the default value is `null`.

Initialization to default values is typically done by having the memory manager or garbage collector initialize memory to all-bits-zero before it is allocated for use. For this reason, it is convenient to use all-bits-zero to represent the null reference.

Definite assignment

At a given location in the executable code of a function member, a variable is said to be *definitely assigned* if the compiler can prove, by a particular static flow analysis ([Precise rules for determining definite assignment](#)), that the variable has been automatically initialized or has been the target of at least one assignment. Informally stated, the rules of definite assignment are:

- An initially assigned variable ([Initially assigned variables](#)) is always considered definitely assigned.
- An initially unassigned variable ([Initially unassigned variables](#)) is considered definitely assigned at a given location if all possible execution paths leading to that location contain at least one of the following:
 - A simple assignment ([Simple assignment](#)) in which the variable is the left operand.
 - An invocation expression ([Invocation expressions](#)) or object creation expression ([Object creation expressions](#)) that passes the variable as an output parameter.
 - For a local variable, a local variable declaration ([Local variable declarations](#)) that includes a variable initializer.

The formal specification underlying the above informal rules is described in [Initially assigned variables](#), [Initially unassigned variables](#), and [Precise rules for determining definite assignment](#).

The definite assignment states of instance variables of a *struct_type* variable are tracked individually as well as collectively. In addition to the rules above, the following rules apply to *struct_type* variables and their instance variables:

- An instance variable is considered definitely assigned if its containing *struct_type* variable is considered definitely assigned.
- A *struct_type* variable is considered definitely assigned if each of its instance variables is considered definitely assigned.

Definite assignment is a requirement in the following contexts:

- A variable must be definitely assigned at each location where its value is obtained. This ensures that undefined values never occur. The occurrence of a variable in an expression is considered to obtain the value of the variable, except when
 - the variable is the left operand of a simple assignment,
 - the variable is passed as an output parameter, or
 - the variable is a *struct_type* variable and occurs as the left operand of a member access.
- A variable must be definitely assigned at each location where it is passed as a reference parameter. This ensures that the function member being invoked can consider the reference parameter initially assigned.
- All output parameters of a function member must be definitely assigned at each location where the function

member returns (through a `return` statement or through execution reaching the end of the function member body). This ensures that function members do not return undefined values in output parameters, thus enabling the compiler to consider a function member invocation that takes a variable as an output parameter equivalent to an assignment to the variable.

- The `this` variable of a *struct_type* instance constructor must be definitely assigned at each location where that instance constructor returns.

Initially assigned variables

The following categories of variables are classified as initially assigned:

- Static variables.
- Instance variables of class instances.
- Instance variables of initially assigned struct variables.
- Array elements.
- Value parameters.
- Reference parameters.
- Variables declared in a `catch` clause or a `foreach` statement.

Initially unassigned variables

The following categories of variables are classified as initially unassigned:

- Instance variables of initially unassigned struct variables.
- Output parameters, including the `this` variable of struct instance constructors.
- Local variables, except those declared in a `catch` clause or a `foreach` statement.

Precise rules for determining definite assignment

In order to determine that each used variable is definitely assigned, the compiler must use a process that is equivalent to the one described in this section.

The compiler processes the body of each function member that has one or more initially unassigned variables. For each initially unassigned variable v , the compiler determines a ***definite assignment state*** for v at each of the following points in the function member:

- At the beginning of each statement
- At the end point ([End points and reachability](#)) of each statement
- On each arc which transfers control to another statement or to the end point of a statement
- At the beginning of each expression
- At the end of each expression

The definite assignment state of v can be either:

- Definitely assigned. This indicates that on all possible control flows to this point, v has been assigned a value.
- Not definitely assigned. For the state of a variable at the end of an expression of type `bool`, the state of a variable that isn't definitely assigned may (but doesn't necessarily) fall into one of the following sub-states:
 - Definitely assigned after true expression. This state indicates that v is definitely assigned if the boolean expression evaluated as true, but is not necessarily assigned if the boolean expression evaluated as false.
 - Definitely assigned after false expression. This state indicates that v is definitely assigned if the boolean expression evaluated as false, but is not necessarily assigned if the boolean expression evaluated as true.

The following rules govern how the state of a variable v is determined at each location.

General rules for statements

- v is not definitely assigned at the beginning of a function member body.
- v is definitely assigned at the beginning of any unreachable statement.
- The definite assignment state of v at the beginning of any other statement is determined by checking the definite assignment state of v on all control flow transfers that target the beginning of that statement. If (and only if) v is definitely assigned on all such control flow transfers, then v is definitely assigned at the beginning of the statement. The set of possible control flow transfers is determined in the same way as for checking statement reachability ([End points and reachability](#)).
- The definite assignment state of v at the end point of a block, `checked`, `unchecked`, `if`, `while`, `do`, `for`, `foreach`, `lock`, `using`, or `switch` statement is determined by checking the definite assignment state of v on all control flow transfers that target the end point of that statement. If v is definitely assigned on all such control flow transfers, then v is definitely assigned at the end point of the statement. Otherwise, v is not definitely assigned at the end point of the statement. The set of possible control flow transfers is determined in the same way as for checking statement reachability ([End points and reachability](#)).

Block statements, checked, and unchecked statements

The definite assignment state of v on the control transfer to the first statement of the statement list in the block (or to the end point of the block, if the statement list is empty) is the same as the definite assignment statement of v before the block, `checked`, or `unchecked` statement.

Expression statements

For an expression statement *stmt* that consists of the expression *expr*:

- v has the same definite assignment state at the beginning of *expr* as at the beginning of *stmt*.
- If v is definitely assigned at the end of *expr*, it is definitely assigned at the end point of *stmt*; otherwise, it is not definitely assigned at the end point of *stmt*.

Declaration statements

- If *stmt* is a declaration statement without initializers, then v has the same definite assignment state at the end point of *stmt* as at the beginning of *stmt*.
- If *stmt* is a declaration statement with initializers, then the definite assignment state for v is determined as if *stmt* were a statement list, with one assignment statement for each declaration with an initializer (in the order of declaration).

If statements

For an `if` statement *stmt* of the form:

```
if ( expr ) then_stmt else else_stmt
```

- v has the same definite assignment state at the beginning of *expr* as at the beginning of *stmt*.
- If v is definitely assigned at the end of *expr*, then it is definitely assigned on the control flow transfer to *then_stmt* and to either *else_stmt* or to the end-point of *stmt* if there is no else clause.
- If v has the state "definitely assigned after true expression" at the end of *expr*, then it is definitely assigned on the control flow transfer to *then_stmt*, and not definitely assigned on the control flow transfer to either *else_stmt* or to the end-point of *stmt* if there is no else clause.
- If v has the state "definitely assigned after false expression" at the end of *expr*, then it is definitely assigned on the control flow transfer to *else_stmt*, and not definitely assigned on the control flow transfer to *then_stmt*. It is definitely assigned at the end-point of *stmt* if and only if it is definitely assigned at the end-point of *then_stmt*.
- Otherwise, v is considered not definitely assigned on the control flow transfer to either the *then_stmt* or *else_stmt*, or to the end-point of *stmt* if there is no else clause.

Switch statements

In a `switch` statement *stmt* with a controlling expression *expr*:

- The definite assignment state of v at the beginning of $expr$ is the same as the state of v at the beginning of $stmt$.
- The definite assignment state of v on the control flow transfer to a reachable switch block statement list is the same as the definite assignment state of v at the end of $expr$.

While statements

For a `while` statement $stmt$ of the form:

```
while ( expr ) while_body
```

- v has the same definite assignment state at the beginning of $expr$ as at the beginning of $stmt$.
- If v is definitely assigned at the end of $expr$, then it is definitely assigned on the control flow transfer to $while_body$ and to the end point of $stmt$.
- If v has the state "definitely assigned after true expression" at the end of $expr$, then it is definitely assigned on the control flow transfer to $while_body$, but not definitely assigned at the end-point of $stmt$.
- If v has the state "definitely assigned after false expression" at the end of $expr$, then it is definitely assigned on the control flow transfer to the end point of $stmt$, but not definitely assigned on the control flow transfer to $while_body$.

Do statements

For a `do` statement $stmt$ of the form:

```
do do_body while ( expr ) ;
```

- v has the same definite assignment state on the control flow transfer from the beginning of $stmt$ to do_body as at the beginning of $stmt$.
- v has the same definite assignment state at the beginning of $expr$ as at the end point of do_body .
- If v is definitely assigned at the end of $expr$, then it is definitely assigned on the control flow transfer to the end point of $stmt$.
- If v has the state "definitely assigned after false expression" at the end of $expr$, then it is definitely assigned on the control flow transfer to the end point of $stmt$.

For statements

Definite assignment checking for a `for` statement of the form:

```
for ( for_initializer ; for_condition ; for_iterator ) embedded_statement
```

is done as if the statement were written:

```
{
    for_initializer ;
    while ( for_condition ) {
        embedded_statement ;
        for_iterator ;
    }
}
```

If the *for_condition* is omitted from the `for` statement, then evaluation of definite assignment proceeds as if *for_condition* were replaced with `true` in the above expansion.

Break, continue, and goto statements

The definite assignment state of v on the control flow transfer caused by a `break`, `continue`, or `goto` statement is the same as the definite assignment state of v at the beginning of the statement.

Throw statements

For a statement *stmt* of the form

```
throw expr ;
```

The definite assignment state of *v* at the beginning of *expr* is the same as the definite assignment state of *v* at the beginning of *stmt*.

Return statements

For a statement *stmt* of the form

```
return expr ;
```

- The definite assignment state of *v* at the beginning of *expr* is the same as the definite assignment state of *v* at the beginning of *stmt*.
- If *v* is an output parameter, then it must be definitely assigned either:
 - after *expr*
 - or at the end of the `finally` block of a `try - finally` or `try - catch - finally` that encloses the `return` statement.

For a statement *stmt* of the form:

```
return ;
```

- If *v* is an output parameter, then it must be definitely assigned either:
 - before *stmt*
 - or at the end of the `finally` block of a `try - finally` or `try - catch - finally` that encloses the `return` statement.

Try-catch statements

For a statement *stmt* of the form:

```
try try_block  
catch(...) catch_block_1  
...  
catch(...) catch_block_n
```

- The definite assignment state of *v* at the beginning of *try_block* is the same as the definite assignment state of *v* at the beginning of *stmt*.
- The definite assignment state of *v* at the beginning of *catch_block_i* (for any *i*) is the same as the definite assignment state of *v* at the beginning of *stmt*.
- The definite assignment state of *v* at the end-point of *stmt* is definitely assigned if (and only if) *v* is definitely assigned at the end-point of *try_block* and every *catch_block_i* (for every *i* from 1 to *n*).

Try-finally statements

For a `try` statement *stmt* of the form:

```
try try_block finally finally_block
```

- The definite assignment state of *v* at the beginning of *try_block* is the same as the definite assignment state of *v* at the beginning of *stmt*.

- The definite assignment state of v at the beginning of *finally_block* is the same as the definite assignment state of v at the beginning of *stmt*.
- The definite assignment state of v at the end-point of *stmt* is definitely assigned if (and only if) at least one of the following is true:
 - v is definitely assigned at the end-point of *try_block*
 - v is definitely assigned at the end-point of *finally_block*

If a control flow transfer (for example, a `goto` statement) is made that begins within *try_block*, and ends outside of *try_block*, then v is also considered definitely assigned on that control flow transfer if v is definitely assigned at the end-point of *finally_block*. (This is not an only if—if v is definitely assigned for another reason on this control flow transfer, then it is still considered definitely assigned.)

Try-catch-finally statements

Definite assignment analysis for a `try` - `catch` - `finally` statement of the form:

```
try try_block
catch(...) catch_block_1
...
catch(...) catch_block_n
finally *finally_block*
```

is done as if the statement were a `try` - `finally` statement enclosing a `try` - `catch` statement:

```
try {
    try try_block
    catch(...) catch_block_1
    ...
    catch(...) catch_block_n
}
finally finally_block
```

The following example demonstrates how the different blocks of a `try` statement ([The try statement](#)) affect definite assignment.

```

class A
{
    static void F() {
        int i, j;
        try {
            goto LABEL;
            // neither i nor j definitely assigned
            i = 1;
            // i definitely assigned
        }

        catch {
            // neither i nor j definitely assigned
            i = 3;
            // i definitely assigned
        }

        finally {
            // neither i nor j definitely assigned
            j = 5;
            // j definitely assigned
        }
        // i and j definitely assigned
    LABEL:;
        // j definitely assigned
    }
}

```

Foreach statements

For a `foreach` statement *stmt* of the form:

```
foreach ( type identifier in expr ) embedded_statement
```

- The definite assignment state of *v* at the beginning of *expr* is the same as the state of *v* at the beginning of *stmt*.
- The definite assignment state of *v* on the control flow transfer to *embedded_statement* or to the end point of *stmt* is the same as the state of *v* at the end of *expr*.

Using statements

For a `using` statement *stmt* of the form:

```
using ( resource_acquisition ) embedded_statement
```

- The definite assignment state of *v* at the beginning of *resource_acquisition* is the same as the state of *v* at the beginning of *stmt*.
- The definite assignment state of *v* on the control flow transfer to *embedded_statement* is the same as the state of *v* at the end of *resource_acquisition*.

Lock statements

For a `lock` statement *stmt* of the form:

```
lock ( expr ) embedded_statement
```

- The definite assignment state of *v* at the beginning of *expr* is the same as the state of *v* at the beginning of *stmt*.
- The definite assignment state of *v* on the control flow transfer to *embedded_statement* is the same as the

state of v at the end of *expr*.

Yield statements

For a `yield return` statement *stmt* of the form:

```
yield return expr ;
```

- The definite assignment state of v at the beginning of *expr* is the same as the state of v at the beginning of *stmt*.
- The definite assignment state of v at the end of *stmt* is the same as the state of v at the end of *expr*.
- A `yield break` statement has no effect on the definite assignment state.

General rules for simple expressions

The following rule applies to these kinds of expressions: literals ([Literals](#)), simple names ([Simple names](#)), member access expressions ([Member access](#)), non-indexed base access expressions ([Base access](#)), `typeof` expressions ([The typeof operator](#)), default value expressions ([Default value expressions](#)) and `nameof` expressions ([Nameof expressions](#)).

- The definite assignment state of v at the end of such an expression is the same as the definite assignment state of v at the beginning of the expression.

General rules for expressions with embedded expressions

The following rules apply to these kinds of expressions: parenthesized expressions ([Parenthesized expressions](#)), element access expressions ([Element access](#)), base access expressions with indexing ([Base access](#)), increment and decrement expressions ([Postfix increment and decrement operators](#), [Prefix increment and decrement operators](#)), cast expressions ([Cast expressions](#)), unary `+`, `-`, `~`, `*` expressions, binary `+`, `-`, `*`, `/`, `%`, `<<`, `>>`, `<`, `<=`, `>`, `>=`, `==`, `!=`, `is`, `as`, `&`, `|`, `^` expressions ([Arithmetic operators](#), [Shift operators](#), [Relational and type-testing operators](#), [Logical operators](#)), compound assignment expressions ([Compound assignment](#)), `checked` and `unchecked` expressions ([The checked and unchecked operators](#)), plus array and delegate creation expressions ([The new operator](#)).

Each of these expressions has one or more sub-expressions that are unconditionally evaluated in a fixed order. For example, the binary `%` operator evaluates the left hand side of the operator, then the right hand side. An indexing operation evaluates the indexed expression, and then evaluates each of the index expressions, in order from left to right. For an expression *expr*, which has sub-expressions *e1*, *e2*, ..., *eN*, evaluated in that order:

- The definite assignment state of v at the beginning of *e1* is the same as the definite assignment state at the beginning of *expr*.
- The definite assignment state of v at the beginning of *ei* (*i* greater than one) is the same as the definite assignment state at the end of the previous sub-expression.
- The definite assignment state of v at the end of *expr* is the same as the definite assignment state at the end of *eN*.

Invocation expressions and object creation expressions

For an invocation expression *expr* of the form:

```
primary_expression ( arg1 , arg2 , ... , argN )
```

or an object creation expression of the form:

```
new type ( arg1 , arg2 , ... , argN )
```

- For an invocation expression, the definite assignment state of v before *primary_expression* is the same as the

state of v before $expr$.

- For an invocation expression, the definite assignment state of v before $arg1$ is the same as the state of v after $primary_expression$.
- For an object creation expression, the definite assignment state of v before $arg1$ is the same as the state of v before $expr$.
- For each argument arg_i , the definite assignment state of v after arg_i is determined by the normal expression rules, ignoring any `ref` or `out` modifiers.
- For each argument arg_i for any i greater than one, the definite assignment state of v before arg_i is the same as the state of v after the previous arg .
- If the variable v is passed as an `out` argument (i.e., an argument of the form `out v`) in any of the arguments, then the state of v after $expr$ is definitely assigned. Otherwise; the state of v after $expr$ is the same as the state of v after $argN$.
- For array initializers ([Array creation expressions](#)), object initializers ([Object initializers](#)), collection initializers ([Collection initializers](#)) and anonymous object initializers ([Anonymous object creation expressions](#)), the definite assignment state is determined by the expansion that these constructs are defined in terms of.

Simple assignment expressions

For an expression $expr$ of the form `w = expr_rhs`:

- The definite assignment state of v before $expr_rhs$ is the same as the definite assignment state of v before $expr$.
- The definite assignment state of v after $expr$ is determined by:
 - If w is the same variable as v , then the definite assignment state of v after $expr$ is definitely assigned.
 - Otherwise, if the assignment occurs within the instance constructor of a struct type, if w is a property access designating an automatically implemented property P on the instance being constructed and v is the hidden backing field of P , then the definite assignment state of v after $expr$ is definitely assigned.
 - Otherwise, the definite assignment state of v after $expr$ is the same as the definite assignment state of v after $expr_rhs$.

&& (conditional AND) expressions

For an expression $expr$ of the form `expr_first && expr_second`:

- The definite assignment state of v before $expr_first$ is the same as the definite assignment state of v before $expr$.
- The definite assignment state of v before $expr_second$ is definitely assigned if the state of v after $expr_first$ is either definitely assigned or "definitely assigned after true expression". Otherwise, it is not definitely assigned.
- The definite assignment state of v after $expr$ is determined by:
 - If $expr_first$ is a constant expression with the value `false`, then the definite assignment state of v after $expr$ is the same as the definite assignment state of v after $expr_first$.
 - Otherwise, if the state of v after $expr_first$ is definitely assigned, then the state of v after $expr$ is definitely assigned.
 - Otherwise, if the state of v after $expr_second$ is definitely assigned, and the state of v after $expr_first$ is "definitely assigned after false expression", then the state of v after $expr$ is definitely assigned.
 - Otherwise, if the state of v after $expr_second$ is definitely assigned or "definitely assigned after true expression", then the state of v after $expr$ is "definitely assigned after true expression".
 - Otherwise, if the state of v after $expr_first$ is "definitely assigned after false expression", and the state of v after $expr_second$ is "definitely assigned after false expression", then the state of v after $expr$ is "definitely assigned after false expression".
 - Otherwise, the state of v after $expr$ is not definitely assigned.

In the example

```

class A
{
    static void F(int x, int y) {
        int i;
        if (x >= 0 && (i = y) >= 0) {
            // i definitely assigned
        }
        else {
            // i not definitely assigned
        }
        // i not definitely assigned
    }
}

```

the variable `i` is considered definitely assigned in one of the embedded statements of an `if` statement but not in the other. In the `if` statement in method `F`, the variable `i` is definitely assigned in the first embedded statement because execution of the expression `(i = y)` always precedes execution of this embedded statement. In contrast, the variable `i` is not definitely assigned in the second embedded statement, since `x >= 0` might have tested false, resulting in the variable `i` being unassigned.

|| (conditional OR) expressions

For an expression `expr` of the form `expr_first || expr_second`:

- The definite assignment state of `v` before `expr_first` is the same as the definite assignment state of `v` before `expr`.
- The definite assignment state of `v` before `expr_second` is definitely assigned if the state of `v` after `expr_first` is either definitely assigned or "definitely assigned after false expression". Otherwise, it is not definitely assigned.
- The definite assignment statement of `v` after `expr` is determined by:
 - If `expr_first` is a constant expression with the value `true`, then the definite assignment state of `v` after `expr` is the same as the definite assignment state of `v` after `expr_first`.
 - Otherwise, if the state of `v` after `expr_first` is definitely assigned, then the state of `v` after `expr` is definitely assigned.
 - Otherwise, if the state of `v` after `expr_second` is definitely assigned, and the state of `v` after `expr_first` is "definitely assigned after true expression", then the state of `v` after `expr` is definitely assigned.
 - Otherwise, if the state of `v` after `expr_second` is definitely assigned or "definitely assigned after false expression", then the state of `v` after `expr` is "definitely assigned after false expression".
 - Otherwise, if the state of `v` after `expr_first` is "definitely assigned after true expression", and the state of `v` after `expr_second` is "definitely assigned after true expression", then the state of `v` after `expr` is "definitely assigned after true expression".
 - Otherwise, the state of `v` after `expr` is not definitely assigned.

In the example


```

class A
{
    static void G(int x, int y) {
        int i;
        if (x >= 0 || (i = y) >= 0) {
            // i not definitely assigned
        }
        else {
            // i definitely assigned
        }
        // i not definitely assigned
    }
}

```

the variable `i` is considered definitely assigned in one of the embedded statements of an `if` statement but not in the other. In the `if` statement in method `G`, the variable `i` is definitely assigned in the second embedded statement because execution of the expression `(i = y)` always precedes execution of this embedded statement. In contrast, the variable `i` is not definitely assigned in the first embedded statement, since `x >= 0` might have tested true, resulting in the variable `i` being unassigned.

! (logical negation) expressions

For an expression `expr` of the form `! expr_operand`:

- The definite assignment state of `v` before `expr_operand` is the same as the definite assignment state of `v` before `expr`.
- The definite assignment state of `v` after `expr` is determined by:
 - If the state of `v` after `*expr_operand` is definitely assigned, then the state of `v` after `expr` is definitely assigned.
 - If the state of `v` after `*expr_operand` is not definitely assigned, then the state of `v` after `expr` is not definitely assigned.
 - If the state of `v` after `*expr_operand` is "definitely assigned after false expression", then the state of `v` after `expr` is "definitely assigned after true expression".
 - If the state of `v` after `*expr_operand` is "definitely assigned after true expression", then the state of `v` after `expr` is "definitely assigned after false expression".

?? (null coalescing) expressions

For an expression `expr` of the form `expr_first ?? expr_second`:

- The definite assignment state of `v` before `expr_first` is the same as the definite assignment state of `v` before `expr`.
- The definite assignment state of `v` before `expr_second` is the same as the definite assignment state of `v` after `expr_first`.
- The definite assignment statement of `v` after `expr` is determined by:
 - If `expr_first` is a constant expression (Constant expressions) with value null, then the state of `v` after `expr` is the same as the state of `v` after `expr_second`.
- Otherwise, the state of `v` after `expr` is the same as the definite assignment state of `v` after `expr_first`.

?: (conditional) expressions

For an expression `expr` of the form `expr_cond ? expr_true : expr_false`:

- The definite assignment state of `v` before `expr_cond` is the same as the state of `v` before `expr`.
- The definite assignment state of `v` before `expr_true` is definitely assigned if and only if one of the following holds:
 - `expr_cond` is a constant expression with the value `false`
 - the state of `v` after `expr_cond` is definitely assigned or "definitely assigned after true expression".

- The definite assignment state of v before $expr_false$ is definitely assigned if and only if one of the following holds:
 - $expr_cond$ is a constant expression with the value `true`
 - the state of v after $expr_cond$ is definitely assigned or "definitely assigned after false expression".
- The definite assignment state of v after $expr$ is determined by:
 - If $expr_cond$ is a constant expression ([Constant expressions](#)) with value `true` then the state of v after $expr$ is the same as the state of v after $expr_true$.
 - Otherwise, if $expr_cond$ is a constant expression ([Constant expressions](#)) with value `false` then the state of v after $expr$ is the same as the state of v after $expr_false$.
 - Otherwise, if the state of v after $expr_true$ is definitely assigned and the state of v after $expr_false$ is definitely assigned, then the state of v after $expr$ is definitely assigned.
 - Otherwise, the state of v after $expr$ is not definitely assigned.

Anonymous functions

For a *lambda_expression* or *anonymous_method_expression* $expr$ with a body (either *block* or *expression*) $body$:

- The definite assignment state of an outer variable v before $body$ is the same as the state of v before $expr$. That is, definite assignment state of outer variables is inherited from the context of the anonymous function.
- The definite assignment state of an outer variable v after $expr$ is the same as the state of v before $expr$.

The example

```
delegate bool Filter(int i);

void F() {
    int max;

    // Error, max is not definitely assigned
    Filter f = (int n) => n < max;

    max = 5;
    DoWork(f);
}
```

generates a compile-time error since `max` is not definitely assigned where the anonymous function is declared.

The example

```
delegate void D();

void F() {
    int n;
    D d = () => { n = 1; };

    d();

    // Error, n is not definitely assigned
    Console.WriteLine(n);
}
```

also generates a compile-time error since the assignment to `n` in the anonymous function has no effect on the definite assignment state of `n` outside the anonymous function.

Variable references

A *variable_reference* is an *expression* that is classified as a variable. A *variable_reference* denotes a storage location that can be accessed both to fetch the current value and to store a new value.

```
variable_reference  
    : expression  
    ;
```

In C and C++, a *variable_reference* is known as an *lvalue*.

Atomicity of variable references

Reads and writes of the following data types are atomic: `bool`, `char`, `byte`, `sbyte`, `short`, `ushort`, `uint`, `int`, `float`, and reference types. In addition, reads and writes of enum types with an underlying type in the previous list are also atomic. Reads and writes of other types, including `long`, `ulong`, `double`, and `decimal`, as well as user-defined types, are not guaranteed to be atomic. Aside from the library functions designed for that purpose, there is no guarantee of atomic read-modify-write, such as in the case of increment or decrement.

Conversions

12/28/2021 • 47 minutes to read • [Edit Online](#)

A **conversion** enables an expression to be treated as being of a particular type. A conversion may cause an expression of a given type to be treated as having a different type, or it may cause an expression without a type to get a type. Conversions can be *implicit* or *explicit*, and this determines whether an explicit cast is required. For instance, the conversion from type `int` to type `long` is implicit, so expressions of type `int` can implicitly be treated as type `long`. The opposite conversion, from type `long` to type `int`, is explicit and so an explicit cast is required.

```
int a = 123;
long b = a;      // implicit conversion from int to long
int c = (int) b; // explicit conversion from long to int
```

Some conversions are defined by the language. Programs may also define their own conversions ([User-defined conversions](#)).

Implicit conversions

The following conversions are classified as implicit conversions:

- Identity conversions
- Implicit numeric conversions
- Implicit enumeration conversions
- Implicit interpolated string conversions
- Implicit nullable conversions
- Null literal conversions
- Implicit reference conversions
- Boxing conversions
- Implicit dynamic conversions
- Implicit constant expression conversions
- User-defined implicit conversions
- Anonymous function conversions
- Method group conversions

Implicit conversions can occur in a variety of situations, including function member invocations ([Compile-time checking of dynamic overload resolution](#)), cast expressions ([Cast expressions](#)), and assignments ([Assignment operators](#)).

The pre-defined implicit conversions always succeed and never cause exceptions to be thrown. Properly designed user-defined implicit conversions should exhibit these characteristics as well.

For the purposes of conversion, the types `object` and `dynamic` are considered equivalent.

However, dynamic conversions ([Implicit dynamic conversions](#) and [Explicit dynamic conversions](#)) apply only to expressions of type `dynamic` ([The dynamic type](#)).

Identity conversion

An identity conversion converts from any type to the same type. This conversion exists such that an entity that

already has a required type can be said to be convertible to that type.

- Because `object` and `dynamic` are considered equivalent there is an identity conversion between `object` and `dynamic`, and between constructed types that are the same when replacing all occurrences of `dynamic` with `object`.

Implicit numeric conversions

The implicit numeric conversions are:

- From `sbyte` to `short`, `int`, `long`, `float`, `double`, Or `decimal`.
- From `byte` to `short`, `ushort`, `int`, `uint`, `long`, `ulong`, `float`, `double`, Or `decimal`.
- From `short` to `int`, `long`, `float`, `double`, Or `decimal`.
- From `ushort` to `int`, `uint`, `long`, `ulong`, `float`, `double`, Or `decimal`.
- From `int` to `long`, `float`, `double`, Or `decimal`.
- From `uint` to `long`, `ulong`, `float`, `double`, Or `decimal`.
- From `long` to `float`, `double`, Or `decimal`.
- From `ulong` to `float`, `double`, Or `decimal`.
- From `char` to `ushort`, `int`, `uint`, `long`, `ulong`, `float`, `double`, Or `decimal`.
- From `float` to `double`.

Conversions from `int`, `uint`, `long`, or `ulong` to `float` and from `long` or `ulong` to `double` may cause a loss of precision, but will never cause a loss of magnitude. The other implicit numeric conversions never lose any information.

There are no implicit conversions to the `char` type, so values of the other integral types do not automatically convert to the `char` type.

Implicit enumeration conversions

An implicit enumeration conversion permits the *decimal_integer_literal* `0` to be converted to any *enum_type* and to any *nullable_type* whose underlying type is an *enum_type*. In the latter case the conversion is evaluated by converting to the underlying *enum_type* and wrapping the result ([Nullable types](#)).

Implicit interpolated string conversions

An implicit interpolated string conversion permits an *interpolated_string_expression* ([Interpolated strings](#)) to be converted to `System.IFormattable` Or `System.FormattableString` (which implements `System.IFormattable`).

When this conversion is applied a string value is not composed from the interpolated string. Instead an instance of `System.FormattableString` is created, as further described in [Interpolated strings](#).

Implicit nullable conversions

Predefined implicit conversions that operate on non-nullable value types can also be used with nullable forms of those types. For each of the predefined implicit identity and numeric conversions that convert from a non-nullable value type `S` to a non-nullable value type `T`, the following implicit nullable conversions exist:

- An implicit conversion from `S?` to `T?`.
- An implicit conversion from `S` to `T?`.

Evaluation of an implicit nullable conversion based on an underlying conversion from `S` to `T` proceeds as follows:

- If the nullable conversion is from `S?` to `T?`:
 - If the source value is null (`HasValue` property is false), the result is the null value of type `T?`.
 - Otherwise, the conversion is evaluated as an unwrapping from `S?` to `S`, followed by the underlying

conversion from `S` to `T`, followed by a wrapping ([Nullable types](#)) from `T` to `T?`.

- If the nullable conversion is from `S` to `T?`, the conversion is evaluated as the underlying conversion from `S` to `T` followed by a wrapping from `T` to `T?`.

Null literal conversions

An implicit conversion exists from the `null` literal to any nullable type. This conversion produces the null value ([Nullable types](#)) of the given nullable type.

Implicit reference conversions

The implicit reference conversions are:

- From any *reference_type* to `object` and `dynamic`.
- From any *class_type* `S` to any *class_type* `T`, provided `S` is derived from `T`.
- From any *class_type* `S` to any *interface_type* `T`, provided `S` implements `T`.
- From any *interface_type* `S` to any *interface_type* `T`, provided `S` is derived from `T`.
- From an *array_type* `S` with an element type `SE` to an *array_type* `T` with an element type `TE`, provided all of the following are true:
 - `S` and `T` differ only in element type. In other words, `S` and `T` have the same number of dimensions.
 - Both `SE` and `TE` are *reference_types*.
 - An implicit reference conversion exists from `SE` to `TE`.
- From any *array_type* to `System.Array` and the interfaces it implements.
- From a single-dimensional array type `S[]` to `System.Collections.Generic.ICollection<T>` and its base interfaces, provided that there is an implicit identity or reference conversion from `S` to `T`.
- From any *delegate_type* to `System.Delegate` and the interfaces it implements.
- From the null literal to any *reference_type*.
- From any *reference_type* to a *reference_type* `T` if it has an implicit identity or reference conversion to a *reference_type* `T0` and `T0` has an identity conversion to `T`.
- From any *reference_type* to an interface or delegate type `T` if it has an implicit identity or reference conversion to an interface or delegate type `T0` and `T0` is variance-convertible ([Variance conversion](#)) to `T`.
- Implicit conversions involving type parameters that are known to be reference types. See [Implicit conversions involving type parameters](#) for more details on implicit conversions involving type parameters.

The implicit reference conversions are those conversions between *reference_types* that can be proven to always succeed, and therefore require no checks at run-time.

Reference conversions, implicit or explicit, never change the referential identity of the object being converted. In other words, while a reference conversion may change the type of the reference, it never changes the type or value of the object being referred to.

Boxing conversions

A boxing conversion permits a *value_type* to be implicitly converted to a reference type. A boxing conversion exists from any *non_nullable_value_type* to `object` and `dynamic`, to `System.ValueType` and to any *interface_type* implemented by the *non_nullable_value_type*. Furthermore an *enum_type* can be converted to the type `System.Enum`.

A boxing conversion exists from a *nullable_type* to a reference type, if and only if a boxing conversion exists from the underlying *non_nullable_value_type* to the reference type.

A value type has a boxing conversion to an interface type `I` if it has a boxing conversion to an interface type `I0` and `I0` has an identity conversion to `I`.

A value type has a boxing conversion to an interface type `I` if it has a boxing conversion to an interface or delegate type `IO` and `IO` is variance-convertible ([Variance conversion](#)) to `I`.

Boxing a value of a *non_nullable_value_type* consists of allocating an object instance and copying the *value_type* value into that instance. A struct can be boxed to the type `System.ValueType`, since that is a base class for all structs ([Inheritance](#)).

Boxing a value of a *nullable_type* proceeds as follows:

- If the source value is null (`HasValue` property is false), the result is a null reference of the target type.
- Otherwise, the result is a reference to a boxed `T` produced by unwrapping and boxing the source value.

Boxing conversions are described further in [Boxing conversions](#).

Implicit dynamic conversions

An implicit dynamic conversion exists from an expression of type `dynamic` to any type `T`. The conversion is dynamically bound ([Dynamic binding](#)), which means that an implicit conversion will be sought at run-time from the run-time type of the expression to `T`. If no conversion is found, a run-time exception is thrown.

Note that this implicit conversion seemingly violates the advice in the beginning of [Implicit conversions](#) that an implicit conversion should never cause an exception. However it is not the conversion itself, but the *finding* of the conversion that causes the exception. The risk of run-time exceptions is inherent in the use of dynamic binding. If dynamic binding of the conversion is not desired, the expression can be first converted to `object`, and then to the desired type.

The following example illustrates implicit dynamic conversions:

```
object o = "object";
dynamic d = "dynamic";

string s1 = o; // Fails at compile-time -- no conversion exists
string s2 = d; // Compiles and succeeds at run-time
int i = d; // Compiles but fails at run-time -- no conversion exists
```

The assignments to `s2` and `i` both employ implicit dynamic conversions, where the binding of the operations is suspended until run-time. At run-time, implicit conversions are sought from the run-time type of `d` -- `string` -- to the target type. A conversion is found to `string` but not to `int`.

Implicit constant expression conversions

An implicit constant expression conversion permits the following conversions:

- A *constant_expression* ([Constant expressions](#)) of type `int` can be converted to type `sbyte`, `byte`, `short`, `ushort`, `uint`, or `ulong`, provided the value of the *constant_expression* is within the range of the destination type.
- A *constant_expression* of type `long` can be converted to type `ulong`, provided the value of the *constant_expression* is not negative.

Implicit conversions involving type parameters

The following implicit conversions exist for a given type parameter `T`:

- From `T` to its effective base class `C`, from `T` to any base class of `C`, and from `T` to any interface implemented by `C`. At run-time, if `T` is a value type, the conversion is executed as a boxing conversion. Otherwise, the conversion is executed as an implicit reference conversion or identity conversion.
- From `T` to an interface type `I` in `T`'s effective interface set and from `T` to any base interface of `I`. At run-time, if `T` is a value type, the conversion is executed as a boxing conversion. Otherwise, the conversion is executed as an implicit reference conversion or identity conversion.

- From `T` to a type parameter `U`, provided `T` depends on `U` ([Type parameter constraints](#)). At run-time, if `U` is a value type, then `T` and `U` are necessarily the same type and no conversion is performed. Otherwise, if `T` is a value type, the conversion is executed as a boxing conversion. Otherwise, the conversion is executed as an implicit reference conversion or identity conversion.
- From the null literal to `T`, provided `T` is known to be a reference type.
- From `T` to a reference type `I` if it has an implicit conversion to a reference type `S0` and `S0` has an identity conversion to `I`. At run-time the conversion is executed the same way as the conversion to `S0`.
- From `T` to an interface type `I` if it has an implicit conversion to an interface or delegate type `I0` and `I0` is variance-convertible to `I` ([Variance conversion](#)). At run-time, if `T` is a value type, the conversion is executed as a boxing conversion. Otherwise, the conversion is executed as an implicit reference conversion or identity conversion.

If `T` is known to be a reference type ([Type parameter constraints](#)), the conversions above are all classified as implicit reference conversions ([Implicit reference conversions](#)). If `T` is not known to be a reference type, the conversions above are classified as boxing conversions ([Boxing conversions](#)).

User-defined implicit conversions

A user-defined implicit conversion consists of an optional standard implicit conversion, followed by execution of a user-defined implicit conversion operator, followed by another optional standard implicit conversion. The exact rules for evaluating user-defined implicit conversions are described in [Processing of user-defined implicit conversions](#).

Anonymous function conversions and method group conversions

Anonymous functions and method groups do not have types in and of themselves, but may be implicitly converted to delegate types or expression tree types. Anonymous function conversions are described in more detail in [Anonymous function conversions](#) and method group conversions in [Method group conversions](#).

Explicit conversions

The following conversions are classified as explicit conversions:

- All implicit conversions.
- Explicit numeric conversions.
- Explicit enumeration conversions.
- Explicit nullable conversions.
- Explicit reference conversions.
- Explicit interface conversions.
- Unboxing conversions.
- Explicit dynamic conversions
- User-defined explicit conversions.

Explicit conversions can occur in cast expressions ([Cast expressions](#)).

The set of explicit conversions includes all implicit conversions. This means that redundant cast expressions are allowed.

The explicit conversions that are not implicit conversions are conversions that cannot be proven to always succeed, conversions that are known to possibly lose information, and conversions across domains of types sufficiently different to merit explicit notation.

Explicit numeric conversions

The explicit numeric conversions are the conversions from a *numeric_type* to another *numeric_type* for which an implicit numeric conversion ([Implicit numeric conversions](#)) does not already exist:

- From `sbyte` to `byte`, `ushort`, `uint`, `ulong`, or `char`.
- From `byte` to `sbyte` and `char`.
- From `short` to `sbyte`, `byte`, `ushort`, `uint`, `ulong`, or `char`.
- From `ushort` to `sbyte`, `byte`, `short`, or `char`.
- From `int` to `sbyte`, `byte`, `short`, `ushort`, `uint`, `ulong`, or `char`.
- From `uint` to `sbyte`, `byte`, `short`, `ushort`, `int`, or `char`.
- From `long` to `sbyte`, `byte`, `short`, `ushort`, `int`, `uint`, `ulong`, or `char`.
- From `ulong` to `sbyte`, `byte`, `short`, `ushort`, `int`, `uint`, `long`, or `char`.
- From `char` to `sbyte`, `byte`, or `short`.
- From `float` to `sbyte`, `byte`, `short`, `ushort`, `int`, `uint`, `long`, `ulong`, `char`, or `decimal`.
- From `double` to `sbyte`, `byte`, `short`, `ushort`, `int`, `uint`, `long`, `ulong`, `char`, `float`, or `decimal`.
- From `decimal` to `sbyte`, `byte`, `short`, `ushort`, `int`, `uint`, `long`, `ulong`, `char`, `float`, or `double`.

Because the explicit conversions include all implicit and explicit numeric conversions, it is always possible to convert from any *numeric_type* to any other *numeric_type* using a cast expression ([Cast expressions](#)).

The explicit numeric conversions possibly lose information or possibly cause exceptions to be thrown. An explicit numeric conversion is processed as follows:

- For a conversion from an integral type to another integral type, the processing depends on the overflow checking context ([The checked and unchecked operators](#)) in which the conversion takes place:
 - In a `checked` context, the conversion succeeds if the value of the source operand is within the range of the destination type, but throws a `System.OverflowException` if the value of the source operand is outside the range of the destination type.
 - In an `unchecked` context, the conversion always succeeds, and proceeds as follows.
 - If the source type is larger than the destination type, then the source value is truncated by discarding its "extra" most significant bits. The result is then treated as a value of the destination type.
 - If the source type is smaller than the destination type, then the source value is either sign-extended or zero-extended so that it is the same size as the destination type. Sign-extension is used if the source type is signed; zero-extension is used if the source type is unsigned. The result is then treated as a value of the destination type.
 - If the source type is the same size as the destination type, then the source value is treated as a value of the destination type.
- For a conversion from `decimal` to an integral type, the source value is rounded towards zero to the nearest integral value, and this integral value becomes the result of the conversion. If the resulting integral value is outside the range of the destination type, a `System.OverflowException` is thrown.
- For a conversion from `float` or `double` to an integral type, the processing depends on the overflow checking context ([The checked and unchecked operators](#)) in which the conversion takes place:
 - In a `checked` context, the conversion proceeds as follows:
 - If the value of the operand is NaN or infinite, a `System.OverflowException` is thrown.
 - Otherwise, the source operand is rounded towards zero to the nearest integral value. If this integral value is within the range of the destination type then this value is the result of the conversion.
 - Otherwise, a `System.OverflowException` is thrown.
 - In an `unchecked` context, the conversion always succeeds, and proceeds as follows.
 - If the value of the operand is NaN or infinite, the result of the conversion is an unspecified value of the destination type.
 - Otherwise, the source operand is rounded towards zero to the nearest integral value. If this

integral value is within the range of the destination type then this value is the result of the conversion.

- Otherwise, the result of the conversion is an unspecified value of the destination type.
- For a conversion from `double` to `float`, the `double` value is rounded to the nearest `float` value. If the `double` value is too small to represent as a `float`, the result becomes positive zero or negative zero. If the `double` value is too large to represent as a `float`, the result becomes positive infinity or negative infinity. If the `double` value is NaN, the result is also NaN.
- For a conversion from `float` or `double` to `decimal`, the source value is converted to `decimal` representation and rounded to the nearest number after the 28th decimal place if required ([The decimal type](#)). If the source value is too small to represent as a `decimal`, the result becomes zero. If the source value is NaN, infinity, or too large to represent as a `decimal`, a `System.OverflowException` is thrown.
- For a conversion from `decimal` to `float` or `double`, the `decimal` value is rounded to the nearest `double` or `float` value. While this conversion may lose precision, it never causes an exception to be thrown.

Explicit enumeration conversions

The explicit enumeration conversions are:

- From `sbyte`, `byte`, `short`, `ushort`, `int`, `uint`, `long`, `ulong`, `char`, `float`, `double`, or `decimal` to any *enum_type*.
- From any *enum_type* to `sbyte`, `byte`, `short`, `ushort`, `int`, `uint`, `long`, `ulong`, `char`, `float`, `double`, or `decimal`.
- From any *enum_type* to any other *enum_type*.

An explicit enumeration conversion between two types is processed by treating any participating *enum_type* as the underlying type of that *enum_type*, and then performing an implicit or explicit numeric conversion between the resulting types. For example, given an *enum_type* `E` with an underlying type of `int`, a conversion from `E` to `byte` is processed as an explicit numeric conversion ([Explicit numeric conversions](#)) from `int` to `byte`, and a conversion from `byte` to `E` is processed as an implicit numeric conversion ([Implicit numeric conversions](#)) from `byte` to `int`.

Explicit nullable conversions

Explicit nullable conversions permit predefined explicit conversions that operate on non-nullable value types to also be used with nullable forms of those types. For each of the predefined explicit conversions that convert from a non-nullable value type `S` to a non-nullable value type `T` ([Identity conversion](#), [Implicit numeric conversions](#), [Implicit enumeration conversions](#), [Explicit numeric conversions](#), and [Explicit enumeration conversions](#)), the following nullable conversions exist:

- An explicit conversion from `S?` to `T?`.
- An explicit conversion from `S` to `T?`.
- An explicit conversion from `S?` to `T`.

Evaluation of a nullable conversion based on an underlying conversion from `S` to `T` proceeds as follows:

- If the nullable conversion is from `S?` to `T?`:
 - If the source value is null (`HasValue` property is false), the result is the null value of type `T?`.
 - Otherwise, the conversion is evaluated as an unwrapping from `S?` to `S`, followed by the underlying conversion from `S` to `T`, followed by a wrapping from `T` to `T?`.
- If the nullable conversion is from `S` to `T?`, the conversion is evaluated as the underlying conversion from `S` to `T` followed by a wrapping from `T` to `T?`.
- If the nullable conversion is from `S?` to `T`, the conversion is evaluated as an unwrapping from `S?` to `S` followed by the underlying conversion from `S` to `T`.

Note that an attempt to unwrap a nullable value will throw an exception if the value is `null`.

Explicit reference conversions

The explicit reference conversions are:

- From `object` and `dynamic` to any other *reference_type*.
- From any *class_type* `S` to any *class_type* `T`, provided `S` is a base class of `T`.
- From any *class_type* `S` to any *interface_type* `T`, provided `S` is not sealed and provided `S` does not implement `T`.
- From any *interface_type* `S` to any *class_type* `T`, provided `T` is not sealed or provided `T` implements `S`.
- From any *interface_type* `S` to any *interface_type* `T`, provided `S` is not derived from `T`.
- From an *array_type* `S` with an element type `SE` to an *array_type* `T` with an element type `TE`, provided all of the following are true:
 - `S` and `T` differ only in element type. In other words, `S` and `T` have the same number of dimensions.
 - Both `SE` and `TE` are *reference_types*.
 - An explicit reference conversion exists from `SE` to `TE`.
- From `System.Array` and the interfaces it implements to any *array_type*.
- From a single-dimensional array type `S[]` to `System.Collections.Generic.ICollection<T>` and its base interfaces, provided that there is an explicit reference conversion from `S` to `T`.
- From `System.Collections.Generic.ICollection<S>` and its base interfaces to a single-dimensional array type `T[]`, provided that there is an explicit identity or reference conversion from `S` to `T`.
- From `System.Delegate` and the interfaces it implements to any *delegate_type*.
- From a reference type to a reference type `T` if it has an explicit reference conversion to a reference type `T0` and `T0` has an identity conversion `T`.
- From a reference type to an interface or delegate type `T` if it has an explicit reference conversion to an interface or delegate type `T0` and either `T0` is variance-convertible to `T` or `T` is variance-convertible to `T0` ([Variance conversion](#)).
- From `D<S1...Sn>` to `D<T1...Tn>` where `D<X1...Xn>` is a generic delegate type, `D<S1...Sn>` is not compatible with or identical to `D<T1...Tn>`, and for each type parameter `xi` of `D` the following holds:
 - If `xi` is invariant, then `Si` is identical to `Ti`.
 - If `xi` is covariant, then there is an implicit or explicit identity or reference conversion from `Si` to `Ti`.
 - If `xi` is contravariant, then `Si` and `Ti` are either identical or both reference types.
- Explicit conversions involving type parameters that are known to be reference types. For more details on explicit conversions involving type parameters, see [Explicit conversions involving type parameters](#).

The explicit reference conversions are those conversions between reference-types that require run-time checks to ensure they are correct.

For an explicit reference conversion to succeed at run-time, the value of the source operand must be `null`, or the actual type of the object referenced by the source operand must be a type that can be converted to the destination type by an implicit reference conversion ([Implicit reference conversions](#)) or boxing conversion ([Boxing conversions](#)). If an explicit reference conversion fails, a `System.InvalidCastException` is thrown.

Reference conversions, implicit or explicit, never change the referential identity of the object being converted. In other words, while a reference conversion may change the type of the reference, it never changes the type or value of the object being referred to.

Unboxing conversions

An unboxing conversion permits a reference type to be explicitly converted to a *value_type*. An unboxing conversion exists from the types `object`, `dynamic` and `System.ValueType` to any *non_nullable_value_type*, and

from any *interface_type* to any *non_nullable_value_type* that implements the *interface_type*. Furthermore type `System.Enum` can be unboxed to any *enum_type*.

An unboxing conversion exists from a reference type to a *nullable_type* if an unboxing conversion exists from the reference type to the underlying *non_nullable_value_type* of the *nullable_type*.

A value type `S` has an unboxing conversion from an interface type `I` if it has an unboxing conversion from an interface type `I0` and `I0` has an identity conversion to `I`.

A value type `S` has an unboxing conversion from an interface type `I` if it has an unboxing conversion from an interface or delegate type `I0` and either `I0` is variance-convertible to `I` or `I` is variance-convertible to `I0` ([Variance conversion](#)).

An unboxing operation consists of first checking that the object instance is a boxed value of the given *value_type*, and then copying the value out of the instance. Unboxing a null reference to a *nullable_type* produces the null value of the *nullable_type*. A struct can be unboxed from the type `System.ValueType`, since that is a base class for all structs ([Inheritance](#)).

Unboxing conversions are described further in [Unboxing conversions](#).

Explicit dynamic conversions

An explicit dynamic conversion exists from an expression of type `dynamic` to any type `T`. The conversion is dynamically bound ([Dynamic binding](#)), which means that an explicit conversion will be sought at run-time from the run-time type of the expression to `T`. If no conversion is found, a run-time exception is thrown.

If dynamic binding of the conversion is not desired, the expression can be first converted to `object`, and then to the desired type.

Assume the following class is defined:

```
class C
{
    int i;

    public C(int i) { this.i = i; }

    public static explicit operator C(string s)
    {
        return new C(int.Parse(s));
    }
}
```

The following example illustrates explicit dynamic conversions:

```
object o = "1";
dynamic d = "2";

var c1 = (C)o; // Compiles, but explicit reference conversion fails
var c2 = (C)d; // Compiles and user defined conversion succeeds
```

The best conversion of `o` to `C` is found at compile-time to be an explicit reference conversion. This fails at run-time, because `"1"` is not in fact a `C`. The conversion of `d` to `C` however, as an explicit dynamic conversion, is suspended to run-time, where a user defined conversion from the run-time type of `d` -- `string` -- to `C` is found, and succeeds.

Explicit conversions involving type parameters

The following explicit conversions exist for a given type parameter `T`:

- From the effective base class `C` of `T` to `T` and from any base class of `C` to `T`. At run-time, if `T` is a value type, the conversion is executed as an unboxing conversion. Otherwise, the conversion is executed as an explicit reference conversion or identity conversion.
- From any interface type to `T`. At run-time, if `T` is a value type, the conversion is executed as an unboxing conversion. Otherwise, the conversion is executed as an explicit reference conversion or identity conversion.
- From `T` to any *interface_type* `I` provided there is not already an implicit conversion from `T` to `I`. At run-time, if `T` is a value type, the conversion is executed as a boxing conversion followed by an explicit reference conversion. Otherwise, the conversion is executed as an explicit reference conversion or identity conversion.
- From a type parameter `U` to `T`, provided `T` depends on `U` ([Type parameter constraints](#)). At run-time, if `U` is a value type, then `T` and `U` are necessarily the same type and no conversion is performed. Otherwise, if `T` is a value type, the conversion is executed as an unboxing conversion. Otherwise, the conversion is executed as an explicit reference conversion or identity conversion.

If `T` is known to be a reference type, the conversions above are all classified as explicit reference conversions ([Explicit reference conversions](#)). If `T` is not known to be a reference type, the conversions above are classified as unboxing conversions ([Unboxing conversions](#)).

The above rules do not permit a direct explicit conversion from an unconstrained type parameter to a non-interface type, which might be surprising. The reason for this rule is to prevent confusion and make the semantics of such conversions clear. For example, consider the following declaration:

```
class X<T>
{
    public static long F(T t) {
        return (long)t;           // Error
    }
}
```

If the direct explicit conversion of `t` to `int` were permitted, one might easily expect that `X<int>.F(7)` would return `7L`. However, it would not, because the standard numeric conversions are only considered when the types are known to be numeric at binding-time. In order to make the semantics clear, the above example must instead be written:

```
class X<T>
{
    public static long F(T t) {
        return (long)(object)t;    // Ok, but will only work when T is long
    }
}
```

This code will now compile but executing `X<int>.F(7)` would then throw an exception at run-time, since a boxed `int` cannot be converted directly to a `long`.

User-defined explicit conversions

A user-defined explicit conversion consists of an optional standard explicit conversion, followed by execution of a user-defined implicit or explicit conversion operator, followed by another optional standard explicit conversion. The exact rules for evaluating user-defined explicit conversions are described in [Processing of user-defined explicit conversions](#).

Standard conversions

The standard conversions are those pre-defined conversions that can occur as part of a user-defined conversion.

Standard implicit conversions

The following implicit conversions are classified as standard implicit conversions:

- Identity conversions ([Identity conversion](#))
- Implicit numeric conversions ([Implicit numeric conversions](#))
- Implicit nullable conversions ([Implicit nullable conversions](#))
- Implicit reference conversions ([Implicit reference conversions](#))
- Boxing conversions ([Boxing conversions](#))
- Implicit constant expression conversions ([Implicit constant expression conversions](#))
- Implicit conversions involving type parameters ([Implicit conversions involving type parameters](#))

The standard implicit conversions specifically exclude user-defined implicit conversions.

Standard explicit conversions

The standard explicit conversions are all standard implicit conversions plus the subset of the explicit conversions for which an opposite standard implicit conversion exists. In other words, if a standard implicit conversion exists from a type `A` to a type `B`, then a standard explicit conversion exists from type `A` to type `B` and from type `B` to type `A`.

User-defined conversions

C# allows the pre-defined implicit and explicit conversions to be augmented by *user-defined conversions*. User-defined conversions are introduced by declaring conversion operators ([Conversion operators](#)) in class and struct types.

Permitted user-defined conversions

C# permits only certain user-defined conversions to be declared. In particular, it is not possible to redefine an already existing implicit or explicit conversion.

For a given source type `S` and target type `T`, if `S` or `T` are nullable types, let `S0` and `T0` refer to their underlying types, otherwise `S0` and `T0` are equal to `S` and `T` respectively. A class or struct is permitted to declare a conversion from a source type `S` to a target type `T` only if all of the following are true:

- `S0` and `T0` are different types.
- Either `S0` or `T0` is the class or struct type in which the operator declaration takes place.
- Neither `S0` nor `T0` is an *interface type*.
- Excluding user-defined conversions, a conversion does not exist from `S` to `T` or from `T` to `S`.

The restrictions that apply to user-defined conversions are discussed further in [Conversion operators](#).

Lifted conversion operators

Given a user-defined conversion operator that converts from a non-nullable value type `S` to a non-nullable value type `T`, a *lifted conversion operator* exists that converts from `S?` to `T?`. This lifted conversion operator performs an unwrapping from `S?` to `S` followed by the user-defined conversion from `S` to `T` followed by a wrapping from `T` to `T?`, except that a null valued `S?` converts directly to a null valued `T?`.

A lifted conversion operator has the same implicit or explicit classification as its underlying user-defined conversion operator. The term "user-defined conversion" applies to the use of both user-defined and lifted conversion operators.

Evaluation of user-defined conversions

A user-defined conversion converts a value from its type, called the *source type*, to another type, called the *target type*. Evaluation of a user-defined conversion centers on finding the *most specific* user-defined conversion operator for the particular source and target types. This determination is broken into several steps:

- Finding the set of classes and structs from which user-defined conversion operators will be considered. This set consists of the source type and its base classes and the target type and its base classes (with the implicit assumptions that only classes and structs can declare user-defined operators, and that non-class types have no base classes). For the purposes of this step, if either the source or target type is a *nullable_type*, their underlying type is used instead.
- From that set of types, determining which user-defined and lifted conversion operators are applicable. For a conversion operator to be applicable, it must be possible to perform a standard conversion ([Standard conversions](#)) from the source type to the operand type of the operator, and it must be possible to perform a standard conversion from the result type of the operator to the target type.
- From the set of applicable user-defined operators, determining which operator is unambiguously the most specific. In general terms, the most specific operator is the operator whose operand type is "closest" to the source type and whose result type is "closest" to the target type. User-defined conversion operators are preferred over lifted conversion operators. The exact rules for establishing the most specific user-defined conversion operator are defined in the following sections.

Once a most specific user-defined conversion operator has been identified, the actual execution of the user-defined conversion involves up to three steps:

- First, if required, performing a standard conversion from the source type to the operand type of the user-defined or lifted conversion operator.
- Next, invoking the user-defined or lifted conversion operator to perform the conversion.
- Finally, if required, performing a standard conversion from the result type of the user-defined or lifted conversion operator to the target type.

Evaluation of a user-defined conversion never involves more than one user-defined or lifted conversion operator. In other words, a conversion from type `S` to type `T` will never first execute a user-defined conversion from `S` to `X` and then execute a user-defined conversion from `X` to `T`.

Exact definitions of evaluation of user-defined implicit or explicit conversions are given in the following sections. The definitions make use of the following terms:

- If a standard implicit conversion ([Standard implicit conversions](#)) exists from a type `A` to a type `B`, and if neither `A` nor `B` are *interface_types*, then `A` is said to be **encompassed by** `B`, and `B` is said to **encompass** `A`.
- The **most encompassing type** in a set of types is the one type that encompasses all other types in the set. If no single type encompasses all other types, then the set has no most encompassing type. In more intuitive terms, the most encompassing type is the "largest" type in the set—the one type to which each of the other types can be implicitly converted.
- The **most encompassed type** in a set of types is the one type that is encompassed by all other types in the set. If no single type is encompassed by all other types, then the set has no most encompassed type. In more intuitive terms, the most encompassed type is the "smallest" type in the set—the one type that can be implicitly converted to each of the other types.

Processing of user-defined implicit conversions

A user-defined implicit conversion from type `S` to type `T` is processed as follows:

- Determine the types `S0` and `T0`. If `S` or `T` are nullable types, `S0` and `T0` are their underlying types, otherwise `S0` and `T0` are equal to `S` and `T` respectively.
- Find the set of types, `D`, from which user-defined conversion operators will be considered. This set consists of `S0` (if `S0` is a class or struct), the base classes of `S0` (if `S0` is a class), and `T0` (if `T0` is a class or struct).
- Find the set of applicable user-defined and lifted conversion operators, `U`. This set consists of the user-defined and lifted implicit conversion operators declared by the classes or structs in `D` that convert from a type encompassing `S` to a type encompassed by `T`. If `U` is empty, the conversion is undefined and a

compile-time error occurs.

- Find the most specific source type, S_x , of the operators in U :
 - If any of the operators in U convert from S , then S_x is S .
 - Otherwise, S_x is the most encompassed type in the combined set of source types of the operators in U . If exactly one most encompassed type cannot be found, then the conversion is ambiguous and a compile-time error occurs.
- Find the most specific target type, T_x , of the operators in U :
 - If any of the operators in U convert to T , then T_x is T .
 - Otherwise, T_x is the most encompassing type in the combined set of target types of the operators in U . If exactly one most encompassing type cannot be found, then the conversion is ambiguous and a compile-time error occurs.
- Find the most specific conversion operator:
 - If U contains exactly one user-defined conversion operator that converts from S_x to T_x , then this is the most specific conversion operator.
 - Otherwise, if U contains exactly one lifted conversion operator that converts from S_x to T_x , then this is the most specific conversion operator.
 - Otherwise, the conversion is ambiguous and a compile-time error occurs.
- Finally, apply the conversion:
 - If S is not S_x , then a standard implicit conversion from S to S_x is performed.
 - The most specific conversion operator is invoked to convert from S_x to T_x .
 - If T_x is not T , then a standard implicit conversion from T_x to T is performed.

Processing of user-defined explicit conversions

A user-defined explicit conversion from type S to type T is processed as follows:

- Determine the types S_0 and T_0 . If S or T are nullable types, S_0 and T_0 are their underlying types, otherwise S_0 and T_0 are equal to S and T respectively.
- Find the set of types, D , from which user-defined conversion operators will be considered. This set consists of S_0 (if S_0 is a class or struct), the base classes of S_0 (if S_0 is a class), T_0 (if T_0 is a class or struct), and the base classes of T_0 (if T_0 is a class).
- Find the set of applicable user-defined and lifted conversion operators, U . This set consists of the user-defined and lifted implicit or explicit conversion operators declared by the classes or structs in D that convert from a type encompassing or encompassed by S to a type encompassing or encompassed by T . If U is empty, the conversion is undefined and a compile-time error occurs.
- Find the most specific source type, S_x , of the operators in U :
 - If any of the operators in U convert from S , then S_x is S .
 - Otherwise, if any of the operators in U convert from types that encompass S , then S_x is the most encompassed type in the combined set of source types of those operators. If no most encompassed type can be found, then the conversion is ambiguous and a compile-time error occurs.
 - Otherwise, S_x is the most encompassing type in the combined set of source types of the operators in U . If exactly one most encompassing type cannot be found, then the conversion is ambiguous and a compile-time error occurs.
- Find the most specific target type, T_x , of the operators in U :
 - If any of the operators in U convert to T , then T_x is T .
 - Otherwise, if any of the operators in U convert to types that are encompassed by T , then T_x is the most encompassing type in the combined set of target types of those operators. If exactly one most encompassing type cannot be found, then the conversion is ambiguous and a compile-time error occurs.

- Otherwise, `TX` is the most encompassed type in the combined set of target types of the operators in `U`. If no most encompassed type can be found, then the conversion is ambiguous and a compile-time error occurs.
- Find the most specific conversion operator:
 - If `U` contains exactly one user-defined conversion operator that converts from `SX` to `TX`, then this is the most specific conversion operator.
 - Otherwise, if `U` contains exactly one lifted conversion operator that converts from `SX` to `TX`, then this is the most specific conversion operator.
 - Otherwise, the conversion is ambiguous and a compile-time error occurs.
- Finally, apply the conversion:
 - If `S` is not `SX`, then a standard explicit conversion from `S` to `SX` is performed.
 - The most specific user-defined conversion operator is invoked to convert from `SX` to `TX`.
 - If `TX` is not `T`, then a standard explicit conversion from `TX` to `T` is performed.

Anonymous function conversions

An *anonymous_method_expression* or *lambda_expression* is classified as an anonymous function ([Anonymous function expressions](#)). The expression does not have a type but can be implicitly converted to a compatible delegate type or expression tree type. Specifically, an anonymous function `F` is compatible with a delegate type `D` provided:

- If `F` contains an *anonymous_function_signature*, then `D` and `F` have the same number of parameters.
- If `F` does not contain an *anonymous_function_signature*, then `D` may have zero or more parameters of any type, as long as no parameter of `D` has the `out` parameter modifier.
- If `F` has an explicitly typed parameter list, each parameter in `D` has the same type and modifiers as the corresponding parameter in `F`.
- If `F` has an implicitly typed parameter list, `D` has no `ref` or `out` parameters.
- If the body of `F` is an expression, and either `D` has a `void` return type or `F` is `async` and `D` has the return type `Task`, then when each parameter of `F` is given the type of the corresponding parameter in `D`, the body of `F` is a valid expression (wrt [Expressions](#)) that would be permitted as a *statement_expression* ([Expression statements](#)).
- If the body of `F` is a statement block, and either `D` has a `void` return type or `F` is `async` and `D` has the return type `Task`, then when each parameter of `F` is given the type of the corresponding parameter in `D`, the body of `F` is a valid statement block (wrt [Blocks](#)) in which no `return` statement specifies an expression.
- If the body of `F` is an expression, and *either* `F` is non-`async` and `D` has a non-void return type `T`, *or* `F` is `async` and `D` has a return type `Task<T>`, then when each parameter of `F` is given the type of the corresponding parameter in `D`, the body of `F` is a valid expression (wrt [Expressions](#)) that is implicitly convertible to `T`.
- If the body of `F` is a statement block, and *either* `F` is non-`async` and `D` has a non-void return type `T`, *or* `F` is `async` and `D` has a return type `Task<T>`, then when each parameter of `F` is given the type of the corresponding parameter in `D`, the body of `F` is a valid statement block (wrt [Blocks](#)) with a non-reachable end point in which each `return` statement specifies an expression that is implicitly convertible to `T`.

For the purpose of brevity, this section uses the short form for the task types `Task` and `Task<T>` ([Async functions](#)).

A lambda expression `F` is compatible with an expression tree type `Expression<D>` if `F` is compatible with the delegate type `D`. Note that this does not apply to anonymous methods, only lambda expressions.

Certain lambda expressions cannot be converted to expression tree types: Even though the conversion *exists*, it fails at compile-time. This is the case if the lambda expression:

- Has a *block* body
- Contains simple or compound assignment operators
- Contains a dynamically bound expression
- Is *async*

The examples that follow use a generic delegate type `Func<A,R>` which represents a function that takes an argument of type `A` and returns a value of type `R`:

```
delegate R Func<A,R>(A arg);
```

In the assignments

```
Func<int,int> f1 = x => x + 1;           // Ok
Func<int,double> f2 = x => x + 1;       // Ok
Func<double,int> f3 = x => x + 1;       // Error
Func<int, Task<int>> f4 = async x => x + 1; // Ok
```

the parameter and return types of each anonymous function are determined from the type of the variable to which the anonymous function is assigned.

The first assignment successfully converts the anonymous function to the delegate type `Func<int,int>` because, when `x` is given type `int`, `x+1` is a valid expression that is implicitly convertible to type `int`.

Likewise, the second assignment successfully converts the anonymous function to the delegate type `Func<int,double>` because the result of `x+1` (of type `int`) is implicitly convertible to type `double`.

However, the third assignment is a compile-time error because, when `x` is given type `double`, the result of `x+1` (of type `double`) is not implicitly convertible to type `int`.

The fourth assignment successfully converts the anonymous *async* function to the delegate type `Func<int, Task<int>>` because the result of `x+1` (of type `int`) is implicitly convertible to the result type `int` of the task type `Task<int>`.

Anonymous functions may influence overload resolution, and participate in type inference. See [Function members](#) for further details.

Evaluation of anonymous function conversions to delegate types

Conversion of an anonymous function to a delegate type produces a delegate instance which references the anonymous function and the (possibly empty) set of captured outer variables that are active at the time of the evaluation. When the delegate is invoked, the body of the anonymous function is executed. The code in the body is executed using the set of captured outer variables referenced by the delegate.

The invocation list of a delegate produced from an anonymous function contains a single entry. The exact target object and target method of the delegate are unspecified. In particular, it is unspecified whether the target object of the delegate is `null`, the `this` value of the enclosing function member, or some other object.

Conversions of semantically identical anonymous functions with the same (possibly empty) set of captured outer variable instances to the same delegate types are permitted (but not required) to return the same delegate instance. The term *semantically identical* is used here to mean that execution of the anonymous functions will, in all cases, produce the same effects given the same arguments. This rule permits code such as the following to be optimized.

```

delegate double Function(double x);

class Test
{
    static double[] Apply(double[] a, Function f) {
        double[] result = new double[a.Length];
        for (int i = 0; i < a.Length; i++) result[i] = f(a[i]);
        return result;
    }

    static void F(double[] a, double[] b) {
        a = Apply(a, (double x) => Math.Sin(x));
        b = Apply(b, (double y) => Math.Sin(y));
        ...
    }
}

```

Since the two anonymous function delegates have the same (empty) set of captured outer variables, and since the anonymous functions are semantically identical, the compiler is permitted to have the delegates refer to the same target method. Indeed, the compiler is permitted to return the very same delegate instance from both anonymous function expressions.

Evaluation of anonymous function conversions to expression tree types

Conversion of an anonymous function to an expression tree type produces an expression tree ([Expression tree types](#)). More precisely, evaluation of the anonymous function conversion leads to the construction of an object structure that represents the structure of the anonymous function itself. The precise structure of the expression tree, as well as the exact process for creating it, are implementation defined.

Implementation example

This section describes a possible implementation of anonymous function conversions in terms of other C# constructs. The implementation described here is based on the same principles used by the Microsoft C# compiler, but it is by no means a mandated implementation, nor is it the only one possible. It only briefly mentions conversions to expression trees, as their exact semantics are outside the scope of this specification.

The remainder of this section gives several examples of code that contains anonymous functions with different characteristics. For each example, a corresponding translation to code that uses only other C# constructs is provided. In the examples, the identifier `D` is assumed to represent the following delegate type:

```

public delegate void D();

```

The simplest form of an anonymous function is one that captures no outer variables:

```

class Test
{
    static void F() {
        D d = () => { Console.WriteLine("test"); };
    }
}

```

This can be translated to a delegate instantiation that references a compiler generated static method in which the code of the anonymous function is placed:

```
class Test
{
    static void F() {
        D d = new D(__Method1);
    }

    static void __Method1() {
        Console.WriteLine("test");
    }
}
```

In the following example, the anonymous function references instance members of `this`:

```
class Test
{
    int x;

    void F() {
        D d = () => { Console.WriteLine(x); };
    }
}
```

This can be translated to a compiler generated instance method containing the code of the anonymous function:

```
class Test
{
    int x;

    void F() {
        D d = new D(__Method1);
    }

    void __Method1() {
        Console.WriteLine(x);
    }
}
```

In this example, the anonymous function captures a local variable:

```
class Test
{
    void F() {
        int y = 123;
        D d = () => { Console.WriteLine(y); };
    }
}
```

The lifetime of the local variable must now be extended to at least the lifetime of the anonymous function delegate. This can be achieved by "hoisting" the local variable into a field of a compiler generated class. Instantiation of the local variable ([Instantiation of local variables](#)) then corresponds to creating an instance of the compiler generated class, and accessing the local variable corresponds to accessing a field in the instance of the compiler generated class. Furthermore, the anonymous function becomes an instance method of the compiler generated class:

```

class Test
{
    void F() {
        __Locals1 __locals1 = new __Locals1();
        __locals1.y = 123;
        D d = new D(__locals1.__Method1);
    }

    class __Locals1
    {
        public int y;

        public void __Method1() {
            Console.WriteLine(y);
        }
    }
}

```

Finally, the following anonymous function captures `this` as well as two local variables with different lifetimes:

```

class Test
{
    int x;

    void F() {
        int y = 123;
        for (int i = 0; i < 10; i++) {
            int z = i * 2;
            D d = () => { Console.WriteLine(x + y + z); };
        }
    }
}

```

Here, a compiler generated class is created for each statement block in which locals are captured such that the locals in the different blocks can have independent lifetimes. An instance of `__Locals2`, the compiler generated class for the inner statement block, contains the local variable `z` and a field that references an instance of `__Locals1`. An instance of `__Locals1`, the compiler generated class for the outer statement block, contains the local variable `y` and a field that references `this` of the enclosing function member. With these data structures it is possible to reach all captured outer variables through an instance of `__Local2`, and the code of the anonymous function can thus be implemented as an instance method of that class.

```

class Test
{
    void F() {
        __Locals1 __locals1 = new __Locals1();
        __locals1.__this = this;
        __locals1.y = 123;
        for (int i = 0; i < 10; i++) {
            __Locals2 __locals2 = new __Locals2();
            __locals2.__locals1 = __locals1;
            __locals2.z = i * 2;
            D d = new D(__locals2.__Method1);
        }
    }

    class __Locals1
    {
        public Test __this;
        public int y;
    }

    class __Locals2
    {
        public __Locals1 __locals1;
        public int z;

        public void __Method1() {
            Console.WriteLine(__locals1.__this.x + __locals1.y + z);
        }
    }
}

```

The same technique applied here to capture local variables can also be used when converting anonymous functions to expression trees: References to the compiler generated objects can be stored in the expression tree, and access to the local variables can be represented as field accesses on these objects. The advantage of this approach is that it allows the "lifted" local variables to be shared between delegates and expression trees.

Method group conversions

An implicit conversion ([Implicit conversions](#)) exists from a method group ([Expression classifications](#)) to a compatible delegate type. Given a delegate type `D` and an expression `E` that is classified as a method group, an implicit conversion exists from `E` to `D` if `E` contains at least one method that is applicable in its normal form ([Applicable function member](#)) to an argument list constructed by use of the parameter types and modifiers of `D`, as described in the following.

The compile-time application of a conversion from a method group `E` to a delegate type `D` is described in the following. Note that the existence of an implicit conversion from `E` to `D` does not guarantee that the compile-time application of the conversion will succeed without error.

- A single method `M` is selected corresponding to a method invocation ([Method invocations](#)) of the form `E(A)`, with the following modifications:
 - The argument list `A` is a list of expressions, each classified as a variable and with the type and modifier (`ref` or `out`) of the corresponding parameter in the *formal_parameter_list* of `D`.
 - The candidate methods considered are only those methods that are applicable in their normal form ([Applicable function member](#)), not those applicable only in their expanded form.
- If the algorithm of [Method invocations](#) produces an error, then a compile-time error occurs. Otherwise the algorithm produces a single best method `M` having the same number of parameters as `D` and the conversion is considered to exist.
- The selected method `M` must be compatible ([Delegate compatibility](#)) with the delegate type `D`, or

otherwise, a compile-time error occurs.

- If the selected method `M` is an instance method, the instance expression associated with `E` determines the target object of the delegate.
- If the selected method `M` is an extension method which is denoted by means of a member access on an instance expression, that instance expression determines the target object of the delegate.
- The result of the conversion is a value of type `D`, namely a newly created delegate that refers to the selected method and target object.
- Note that this process can lead to the creation of a delegate to an extension method, if the algorithm of [Method invocations](#) fails to find an instance method but succeeds in processing the invocation of `E(A)` as an extension method invocation ([Extension method invocations](#)). A delegate thus created captures the extension method as well as its first argument.

The following example demonstrates method group conversions:

```
delegate string D1(object o);

delegate object D2(string s);

delegate object D3();

delegate string D4(object o, params object[] a);

delegate string D5(int i);

class Test
{
    static string F(object o) {...}

    static void G() {
        D1 d1 = F;           // Ok
        D2 d2 = F;           // Ok
        D3 d3 = F;           // Error -- not applicable
        D4 d4 = F;           // Error -- not applicable in normal form
        D5 d5 = F;           // Error -- applicable but not compatible
    }
}
```

The assignment to `d1` implicitly converts the method group `F` to a value of type `D1`.

The assignment to `d2` shows how it is possible to create a delegate to a method that has less derived (contravariant) parameter types and a more derived (covariant) return type.

The assignment to `d3` shows how no conversion exists if the method is not applicable.

The assignment to `d4` shows how the method must be applicable in its normal form.

The assignment to `d5` shows how parameter and return types of the delegate and method are allowed to differ only for reference types.

As with all other implicit and explicit conversions, the cast operator can be used to explicitly perform a method group conversion. Thus, the example

```
object obj = new EventHandler(myDialog.OkClick);
```

could instead be written

```
object obj = (EventHandler)myDialog.OkClick;
```

Method groups may influence overload resolution, and participate in type inference. See [Function members](#) for further details.

The run-time evaluation of a method group conversion proceeds as follows:

- If the method selected at compile-time is an instance method, or it is an extension method which is accessed as an instance method, the target object of the delegate is determined from the instance expression associated with `E`:
 - The instance expression is evaluated. If this evaluation causes an exception, no further steps are executed.
 - If the instance expression is of a *reference_type*, the value computed by the instance expression becomes the target object. If the selected method is an instance method and the target object is `null`, a `System.NullReferenceException` is thrown and no further steps are executed.
 - If the instance expression is of a *value_type*, a boxing operation ([Boxing conversions](#)) is performed to convert the value to an object, and this object becomes the target object.
- Otherwise the selected method is part of a static method call, and the target object of the delegate is `null`.
- A new instance of the delegate type `D` is allocated. If there is not enough memory available to allocate the new instance, a `System.OutOfMemoryException` is thrown and no further steps are executed.
- The new delegate instance is initialized with a reference to the method that was determined at compile-time and a reference to the target object computed above.

Expressions

12/28/2021 • 227 minutes to read • [Edit Online](#)

An expression is a sequence of operators and operands. This chapter defines the syntax, order of evaluation of operands and operators, and meaning of expressions.

Expression classifications

An expression is classified as one of the following:

- A value. Every value has an associated type.
- A variable. Every variable has an associated type, namely the declared type of the variable.
- A namespace. An expression with this classification can only appear as the left hand side of a *member_access* ([Member access](#)). In any other context, an expression classified as a namespace causes a compile-time error.
- A type. An expression with this classification can only appear as the left hand side of a *member_access* ([Member access](#)), or as an operand for the `as` operator ([The as operator](#)), the `is` operator ([The is operator](#)), or the `typeof` operator ([The typeof operator](#)). In any other context, an expression classified as a type causes a compile-time error.
- A method group, which is a set of overloaded methods resulting from a member lookup ([Member lookup](#)). A method group may have an associated instance expression and an associated type argument list. When an instance method is invoked, the result of evaluating the instance expression becomes the instance represented by `this` ([This access](#)). A method group is permitted in an *invocation_expression* ([Invocation expressions](#)), a *delegate_creation_expression* ([Delegate creation expressions](#)) and as the left hand side of an `is` operator, and can be implicitly converted to a compatible delegate type ([Method group conversions](#)). In any other context, an expression classified as a method group causes a compile-time error.
- A null literal. An expression with this classification can be implicitly converted to a reference type or nullable type.
- An anonymous function. An expression with this classification can be implicitly converted to a compatible delegate type or expression tree type.
- A property access. Every property access has an associated type, namely the type of the property. Furthermore, a property access may have an associated instance expression. When an accessor (the `get` or `set` block) of an instance property access is invoked, the result of evaluating the instance expression becomes the instance represented by `this` ([This access](#)).
- An event access. Every event access has an associated type, namely the type of the event. Furthermore, an event access may have an associated instance expression. An event access may appear as the left hand operand of the `+=` and `-=` operators ([Event assignment](#)). In any other context, an expression classified as an event access causes a compile-time error.
- An indexer access. Every indexer access has an associated type, namely the element type of the indexer. Furthermore, an indexer access has an associated instance expression and an associated argument list. When an accessor (the `get` or `set` block) of an indexer access is invoked, the result of evaluating the instance expression becomes the instance represented by `this` ([This access](#)), and the result of evaluating the argument list becomes the parameter list of the invocation.
- Nothing. This occurs when the expression is an invocation of a method with a return type of `void`. An expression classified as nothing is only valid in the context of a *statement_expression* ([Expression statements](#)).

The final result of an expression is never a namespace, type, method group, or event access. Rather, as noted above, these categories of expressions are intermediate constructs that are only permitted in certain contexts.

A property access or indexer access is always reclassified as a value by performing an invocation of the *get accessor* or the *set accessor*. The particular accessor is determined by the context of the property or indexer access: If the access is the target of an assignment, the *set accessor* is invoked to assign a new value ([Simple assignment](#)). Otherwise, the *get accessor* is invoked to obtain the current value ([Values of expressions](#)).

Values of expressions

Most of the constructs that involve an expression ultimately require the expression to denote a **value**. In such cases, if the actual expression denotes a namespace, a type, a method group, or nothing, a compile-time error occurs. However, if the expression denotes a property access, an indexer access, or a variable, the value of the property, indexer, or variable is implicitly substituted:

- The value of a variable is simply the value currently stored in the storage location identified by the variable. A variable must be considered definitely assigned ([Definite assignment](#)) before its value can be obtained, or otherwise a compile-time error occurs.
- The value of a property access expression is obtained by invoking the *get accessor* of the property. If the property has no *get accessor*, a compile-time error occurs. Otherwise, a function member invocation ([Compile-time checking of dynamic overload resolution](#)) is performed, and the result of the invocation becomes the value of the property access expression.
- The value of an indexer access expression is obtained by invoking the *get accessor* of the indexer. If the indexer has no *get accessor*, a compile-time error occurs. Otherwise, a function member invocation ([Compile-time checking of dynamic overload resolution](#)) is performed with the argument list associated with the indexer access expression, and the result of the invocation becomes the value of the indexer access expression.

Static and Dynamic Binding

The process of determining the meaning of an operation based on the type or value of constituent expressions (arguments, operands, receivers) is often referred to as **binding**. For instance the meaning of a method call is determined based on the type of the receiver and arguments. The meaning of an operator is determined based on the type of its operands.

In C# the meaning of an operation is usually determined at compile-time, based on the compile-time type of its constituent expressions. Likewise, if an expression contains an error, the error is detected and reported by the compiler. This approach is known as **static binding**.

However, if an expression is a dynamic expression (i.e. has the type `dynamic`) this indicates that any binding that it participates in should be based on its run-time type (i.e. the actual type of the object it denotes at run-time) rather than the type it has at compile-time. The binding of such an operation is therefore deferred until the time where the operation is to be executed during the running of the program. This is referred to as **dynamic binding**.

When an operation is dynamically bound, little or no checking is performed by the compiler. Instead if the run-time binding fails, errors are reported as exceptions at run-time.

The following operations in C# are subject to binding:

- Member access: `e.M`
- Method invocation: `e.M(e1, ..., eN)`
- Delegate invocation: `e(e1, ..., eN)`
- Element access: `e[e1, ..., eN]`
- Object creation: `new C(e1, ..., eN)`
- Overloaded unary operators: `+`, `-`, `!`, `~`, `++`, `--`, `true`, `false`
- Overloaded binary operators: `+`, `-`, `*`, `/`, `%`, `&`, `&&`, `|`, `||`, `??`, `^`, `<<`, `>>`, `==`, `!=`, `>`, `<`, `>=`, `<=`

- Assignment operators: `=`, `+=`, `-=`, `*=`, `/=`, `%=`, `&=`, `|=`, `^=`, `<<=`, `>>=`
- Implicit and explicit conversions

When no dynamic expressions are involved, C# defaults to static binding, which means that the compile-time types of constituent expressions are used in the selection process. However, when one of the constituent expressions in the operations listed above is a dynamic expression, the operation is instead dynamically bound.

Binding-time

Static binding takes place at compile-time, whereas dynamic binding takes place at run-time. In the following sections, the term *binding-time* refers to either compile-time or run-time, depending on when the binding takes place.

The following example illustrates the notions of static and dynamic binding and of binding-time:

```
object o = 5;
dynamic d = 5;

Console.WriteLine(5); // static binding to Console.WriteLine(int)
Console.WriteLine(o); // static binding to Console.WriteLine(object)
Console.WriteLine(d); // dynamic binding to Console.WriteLine(int)
```

The first two calls are statically bound: the overload of `Console.WriteLine` is picked based on the compile-time type of their argument. Thus, the binding-time is compile-time.

The third call is dynamically bound: the overload of `Console.WriteLine` is picked based on the run-time type of its argument. This happens because the argument is a dynamic expression -- its compile-time type is `dynamic`. Thus, the binding-time for the third call is run-time.

Dynamic binding

The purpose of dynamic binding is to allow C# programs to interact with *dynamic objects*, i.e. objects that do not follow the normal rules of the C# type system. Dynamic objects may be objects from other programming languages with different types systems, or they may be objects that are programmatically setup to implement their own binding semantics for different operations.

The mechanism by which a dynamic object implements its own semantics is implementation defined. A given interface -- again implementation defined -- is implemented by dynamic objects to signal to the C# run-time that they have special semantics. Thus, whenever operations on a dynamic object are dynamically bound, their own binding semantics, rather than those of C# as specified in this document, take over.

While the purpose of dynamic binding is to allow interoperation with dynamic objects, C# allows dynamic binding on all objects, whether they are dynamic or not. This allows for a smoother integration of dynamic objects, as the results of operations on them may not themselves be dynamic objects, but are still of a type unknown to the programmer at compile-time. Also dynamic binding can help eliminate error-prone reflection-based code even when no objects involved are dynamic objects.

The following sections describe for each construct in the language exactly when dynamic binding is applied, what compile time checking -- if any -- is applied, and what the compile-time result and expression classification is.

Types of constituent expressions

When an operation is statically bound, the type of a constituent expression (e.g. a receiver, an argument, an index or an operand) is always considered to be the compile-time type of that expression.

When an operation is dynamically bound, the type of a constituent expression is determined in different ways depending on the compile-time type of the constituent expression:

- A constituent expression of compile-time type `dynamic` is considered to have the type of the actual value that the expression evaluates to at runtime
- A constituent expression whose compile-time type is a type parameter is considered to have the type which the type parameter is bound to at runtime
- Otherwise the constituent expression is considered to have its compile-time type.

Operators

Expressions are constructed from *operands* and *operators*. The operators of an expression indicate which operations to apply to the operands. Examples of operators include `+`, `-`, `*`, `/`, and `new`. Examples of operands include literals, fields, local variables, and expressions.

There are three kinds of operators:

- Unary operators. The unary operators take one operand and use either prefix notation (such as `--x`) or postfix notation (such as `x++`).
- Binary operators. The binary operators take two operands and all use infix notation (such as `x + y`).
- Ternary operator. Only one ternary operator, `?:`, exists; it takes three operands and uses infix notation (`c ? x : y`).

The order of evaluation of operators in an expression is determined by the *precedence* and *associativity* of the operators ([Operator precedence and associativity](#)).

Operands in an expression are evaluated from left to right. For example, in `F(i) + G(i++) * H(i)`, method `F` is called using the old value of `i`, then method `G` is called with the old value of `i`, and, finally, method `H` is called with the new value of `i`. This is separate from and unrelated to operator precedence.

Certain operators can be *overloaded*. Operator overloading permits user-defined operator implementations to be specified for operations where one or both of the operands are of a user-defined class or struct type ([Operator overloading](#)).

Operator precedence and associativity

When an expression contains multiple operators, the *precedence* of the operators controls the order in which the individual operators are evaluated. For example, the expression `x + y * z` is evaluated as `x + (y * z)` because the `*` operator has higher precedence than the binary `+` operator. The precedence of an operator is established by the definition of its associated grammar production. For example, an *additive_expression* consists of a sequence of *multiplicative_expressions* separated by `+` or `-` operators, thus giving the `+` and `-` operators lower precedence than the `*`, `/`, and `%` operators.

The following table summarizes all operators in order of precedence from highest to lowest:

SECTION	CATEGORY	OPERATORS
Primary expressions	Primary	<code>x.y</code> <code>f(x)</code> <code>a[x]</code> <code>x++</code> <code>x--</code> <code>new</code> <code>typeof</code> <code>default</code> <code>checked</code> <code>unchecked</code> <code>delegate</code>
Unary operators	Unary	<code>+</code> <code>-</code> <code>!</code> <code>~</code> <code>++x</code> <code>--x</code> <code>(T)x</code>
Arithmetic operators	Multiplicative	<code>*</code> <code>/</code> <code>%</code>
Arithmetic operators	Additive	<code>+</code> <code>-</code>
Shift operators	Shift	<code><<</code> <code>>></code>

SECTION	CATEGORY	OPERATORS
Relational and type-testing operators	Relational and type testing	< > <= >= is as
Relational and type-testing operators	Equality	== !=
Logical operators	Logical AND	&
Logical operators	Logical XOR	^
Logical operators	Logical OR	
Conditional logical operators	Conditional AND	&&
Conditional logical operators	Conditional OR	
The null coalescing operator	Null coalescing	??
Conditional operator	Conditional	?:
Assignment operators, Anonymous function expressions	Assignment and lambda expression	= *= /= %= += -= <<= >>= &= ^= = =>

When an operand occurs between two operators with the same precedence, the associativity of the operators controls the order in which the operations are performed:

- Except for the assignment operators and the null coalescing operator, all binary operators are **left-associative**, meaning that operations are performed from left to right. For example, `x + y + z` is evaluated as `(x + y) + z`.
- The assignment operators, the null coalescing operator and the conditional operator (`?:`) are **right-associative**, meaning that operations are performed from right to left. For example, `x = y = z` is evaluated as `x = (y = z)`.

Precedence and associativity can be controlled using parentheses. For example, `x + y * z` first multiplies `y` by `z` and then adds the result to `x`, but `(x + y) * z` first adds `x` and `y` and then multiplies the result by `z`.

Operator overloading

All unary and binary operators have predefined implementations that are automatically available in any expression. In addition to the predefined implementations, user-defined implementations can be introduced by including `operator` declarations in classes and structs ([Operators](#)). User-defined operator implementations always take precedence over predefined operator implementations: Only when no applicable user-defined operator implementations exist will the predefined operator implementations be considered, as described in [Unary operator overload resolution](#) and [Binary operator overload resolution](#).

The **overloadable unary operators** are:

```
+ - ! ~ ++ -- true false
```

Although `true` and `false` are not used explicitly in expressions (and therefore are not included in the precedence table in [Operator precedence and associativity](#)), they are considered operators because they are

invoked in several expression contexts: boolean expressions ([Boolean expressions](#)) and expressions involving the conditional ([Conditional operator](#)), and conditional logical operators ([Conditional logical operators](#)).

The *overloadable binary operators* are:

+ - * / % & | ^ << >> == != > < >= <=

Only the operators listed above can be overloaded. In particular, it is not possible to overload member access, method invocation, or the `=`, `&&`, `||`, `??`, `?:`, `=>`, `checked`, `unchecked`, `new`, `typeof`, `default`, `as`, and `is` operators.

When a binary operator is overloaded, the corresponding assignment operator, if any, is also implicitly overloaded. For example, an overload of operator `*` is also an overload of operator `*=`. This is described further in [Compound assignment](#). Note that the assignment operator itself (`=`) cannot be overloaded. An assignment always performs a simple bit-wise copy of a value into a variable.

Cast operations, such as `(T)x`, are overloaded by providing user-defined conversions ([User-defined conversions](#)).

Element access, such as `a[x]`, is not considered an overloadable operator. Instead, user-defined indexing is supported through indexers ([Indexers](#)).

In expressions, operators are referenced using operator notation, and in declarations, operators are referenced using functional notation. The following table shows the relationship between operator and functional notations for unary and binary operators. In the first entry, *op* denotes any overloadable unary prefix operator. In the second entry, *op* denotes the unary postfix `++` and `--` operators. In the third entry, *op* denotes any overloadable binary operator.

OPERATOR NOTATION	FUNCTIONAL NOTATION
<code>op x</code>	<code>operator op(x)</code>
<code>x op</code>	<code>operator op(x)</code>
<code>x op y</code>	<code>operator op(x,y)</code>

User-defined operator declarations always require at least one of the parameters to be of the class or struct type that contains the operator declaration. Thus, it is not possible for a user-defined operator to have the same signature as a predefined operator.

User-defined operator declarations cannot modify the syntax, precedence, or associativity of an operator. For example, the `/` operator is always a binary operator, always has the precedence level specified in [Operator precedence and associativity](#), and is always left-associative.

While it is possible for a user-defined operator to perform any computation it pleases, implementations that produce results other than those that are intuitively expected are strongly discouraged. For example, an implementation of `operator ==` should compare the two operands for equality and return an appropriate `bool` result.

The descriptions of individual operators in [Primary expressions](#) through [Conditional logical operators](#) specify the predefined implementations of the operators and any additional rules that apply to each operator. The descriptions make use of the terms *unary operator overload resolution*, *binary operator overload resolution*, and *numeric promotion*, definitions of which are found in the following sections.

Unary operator overload resolution

An operation of the form `op x` or `x op`, where `op` is an overloadable unary operator, and `x` is an expression of type `T`, is processed as follows:

- The set of candidate user-defined operators provided by `x` for the operation `operator op(x)` is determined using the rules of [Candidate user-defined operators](#).
- If the set of candidate user-defined operators is not empty, then this becomes the set of candidate operators for the operation. Otherwise, the predefined unary `operator op` implementations, including their lifted forms, become the set of candidate operators for the operation. The predefined implementations of a given operator are specified in the description of the operator ([Primary expressions](#) and [Unary operators](#)).
- The overload resolution rules of [Overload resolution](#) are applied to the set of candidate operators to select the best operator with respect to the argument list `(x)`, and this operator becomes the result of the overload resolution process. If overload resolution fails to select a single best operator, a binding-time error occurs.

Binary operator overload resolution

An operation of the form `x op y`, where `op` is an overloadable binary operator, `x` is an expression of type `T`, and `y` is an expression of type `U`, is processed as follows:

- The set of candidate user-defined operators provided by `x` and `y` for the operation `operator op(x,y)` is determined. The set consists of the union of the candidate operators provided by `x` and the candidate operators provided by `y`, each determined using the rules of [Candidate user-defined operators](#). If `x` and `y` are the same type, or if `x` and `y` are derived from a common base type, then shared candidate operators only occur in the combined set once.
- If the set of candidate user-defined operators is not empty, then this becomes the set of candidate operators for the operation. Otherwise, the predefined binary `operator op` implementations, including their lifted forms, become the set of candidate operators for the operation. The predefined implementations of a given operator are specified in the description of the operator ([Arithmetic operators](#) through [Conditional logical operators](#)). For predefined enum and delegate operators, the only operators considered are those defined by an enum or delegate type that is the binding-time type of one of the operands.
- The overload resolution rules of [Overload resolution](#) are applied to the set of candidate operators to select the best operator with respect to the argument list `(x,y)`, and this operator becomes the result of the overload resolution process. If overload resolution fails to select a single best operator, a binding-time error occurs.

Candidate user-defined operators

Given a type `T` and an operation `operator op(A)`, where `op` is an overloadable operator and `A` is an argument list, the set of candidate user-defined operators provided by `T` for `operator op(A)` is determined as follows:

- Determine the type `T0`. If `T` is a nullable type, `T0` is its underlying type, otherwise `T0` is equal to `T`.
- For all `operator op` declarations in `T0` and all lifted forms of such operators, if at least one operator is applicable ([Applicable function member](#)) with respect to the argument list `A`, then the set of candidate operators consists of all such applicable operators in `T0`.
- Otherwise, if `T0` is `object`, the set of candidate operators is empty.
- Otherwise, the set of candidate operators provided by `T0` is the set of candidate operators provided by the direct base class of `T0`, or the effective base class of `T0` if `T0` is a type parameter.

Numeric promotions

Numeric promotion consists of automatically performing certain implicit conversions of the operands of the predefined unary and binary numeric operators. Numeric promotion is not a distinct mechanism, but rather an effect of applying overload resolution to the predefined operators. Numeric promotion specifically does not affect evaluation of user-defined operators, although user-defined operators can be implemented to exhibit

similar effects.

As an example of numeric promotion, consider the predefined implementations of the binary `*` operator:

```
int operator *(int x, int y);
uint operator *(uint x, uint y);
long operator *(long x, long y);
ulong operator *(ulong x, ulong y);
float operator *(float x, float y);
double operator *(double x, double y);
decimal operator *(decimal x, decimal y);
```

When overload resolution rules ([Overload resolution](#)) are applied to this set of operators, the effect is to select the first of the operators for which implicit conversions exist from the operand types. For example, for the operation `b * s`, where `b` is a `byte` and `s` is a `short`, overload resolution selects `operator *(int,int)` as the best operator. Thus, the effect is that `b` and `s` are converted to `int`, and the type of the result is `int`. Likewise, for the operation `i * d`, where `i` is an `int` and `d` is a `double`, overload resolution selects `operator *(double,double)` as the best operator.

Unary numeric promotions

Unary numeric promotion occurs for the operands of the predefined `+`, `-`, and `~` unary operators. Unary numeric promotion simply consists of converting operands of type `sbyte`, `byte`, `short`, `ushort`, or `char` to type `int`. Additionally, for the unary `-` operator, unary numeric promotion converts operands of type `uint` to type `long`.

Binary numeric promotions

Binary numeric promotion occurs for the operands of the predefined `+`, `-`, `*`, `/`, `%`, `&`, `|`, `^`, `==`, `!=`, `>`, `<`, `>=`, and `<=` binary operators. Binary numeric promotion implicitly converts both operands to a common type which, in case of the non-relational operators, also becomes the result type of the operation. Binary numeric promotion consists of applying the following rules, in the order they appear here:

- If either operand is of type `decimal`, the other operand is converted to type `decimal`, or a binding-time error occurs if the other operand is of type `float` or `double`.
- Otherwise, if either operand is of type `double`, the other operand is converted to type `double`.
- Otherwise, if either operand is of type `float`, the other operand is converted to type `float`.
- Otherwise, if either operand is of type `ulong`, the other operand is converted to type `ulong`, or a binding-time error occurs if the other operand is of type `sbyte`, `short`, `int`, or `long`.
- Otherwise, if either operand is of type `long`, the other operand is converted to type `long`.
- Otherwise, if either operand is of type `uint` and the other operand is of type `sbyte`, `short`, or `int`, both operands are converted to type `long`.
- Otherwise, if either operand is of type `uint`, the other operand is converted to type `uint`.
- Otherwise, both operands are converted to type `int`.

Note that the first rule disallows any operations that mix the `decimal` type with the `double` and `float` types. The rule follows from the fact that there are no implicit conversions between the `decimal` type and the `double` and `float` types.

Also note that it is not possible for an operand to be of type `ulong` when the other operand is of a signed integral type. The reason is that no integral type exists that can represent the full range of `ulong` as well as the signed integral types.

In both of the above cases, a cast expression can be used to explicitly convert one operand to a type that is compatible with the other operand.

In the example


```
decimal AddPercent(decimal x, double percent) {
    return x * (1.0 + percent / 100.0);
}
```

a binding-time error occurs because a `decimal` cannot be multiplied by a `double`. The error is resolved by explicitly converting the second operand to `decimal`, as follows:

```
decimal AddPercent(decimal x, double percent) {
    return x * (decimal)(1.0 + percent / 100.0);
}
```

Lifted operators

Lifted operators permit predefined and user-defined operators that operate on non-nullable value types to also be used with nullable forms of those types. Lifted operators are constructed from predefined and user-defined operators that meet certain requirements, as described in the following:

- For the unary operators

```
+ ++ - -- ! ~
```

a lifted form of an operator exists if the operand and result types are both non-nullable value types. The lifted form is constructed by adding a single `?` modifier to the operand and result types. The lifted operator produces a null value if the operand is null. Otherwise, the lifted operator unwraps the operand, applies the underlying operator, and wraps the result.

- For the binary operators

```
+ - * / % & | ^ << >>
```

a lifted form of an operator exists if the operand and result types are all non-nullable value types. The lifted form is constructed by adding a single `?` modifier to each operand and result type. The lifted operator produces a null value if one or both operands are null (an exception being the `&` and `|` operators of the `bool?` type, as described in [Boolean logical operators](#)). Otherwise, the lifted operator unwraps the operands, applies the underlying operator, and wraps the result.

- For the equality operators

```
== !=
```

a lifted form of an operator exists if the operand types are both non-nullable value types and if the result type is `bool`. The lifted form is constructed by adding a single `?` modifier to each operand type. The lifted operator considers two null values equal, and a null value unequal to any non-null value. If both operands are non-null, the lifted operator unwraps the operands and applies the underlying operator to produce the `bool` result.

- For the relational operators

```
< > <= >=
```

a lifted form of an operator exists if the operand types are both non-nullable value types and if the result type is `bool`. The lifted form is constructed by adding a single `?` modifier to each operand type. The

lifted operator produces the value `false` if one or both operands are null. Otherwise, the lifted operator unwraps the operands and applies the underlying operator to produce the `bool` result.

Member lookup

A member lookup is the process whereby the meaning of a name in the context of a type is determined. A member lookup can occur as part of evaluating a *simple_name* (Simple names) or a *member_access* (Member access) in an expression. If the *simple_name* or *member_access* occurs as the *primary_expression* of an *invocation_expression* (Method invocations), the member is said to be invoked.

If a member is a method or event, or if it is a constant, field or property of either a delegate type (Delegates) or the type `dynamic` (The dynamic type), then the member is said to be *invocable*.

Member lookup considers not only the name of a member but also the number of type parameters the member has and whether the member is accessible. For the purposes of member lookup, generic methods and nested generic types have the number of type parameters indicated in their respective declarations and all other members have zero type parameters.

A member lookup of a name `N` with `K` type parameters in a type `T` is processed as follows:

- First, a set of accessible members named `N` is determined:
 - If `T` is a type parameter, then the set is the union of the sets of accessible members named `N` in each of the types specified as a primary constraint or secondary constraint (Type parameter constraints) for `T`, along with the set of accessible members named `N` in `object`.
 - Otherwise, the set consists of all accessible (Member access) members named `N` in `T`, including inherited members and the accessible members named `N` in `object`. If `T` is a constructed type, the set of members is obtained by substituting type arguments as described in Members of constructed types. Members that include an `override` modifier are excluded from the set.
- Next, if `K` is zero, all nested types whose declarations include type parameters are removed. If `K` is not zero, all members with a different number of type parameters are removed. Note that when `K` is zero, methods having type parameters are not removed, since the type inference process (Type inference) might be able to infer the type arguments.
- Next, if the member is *invoked*, all non-*invocable* members are removed from the set.
- Next, members that are hidden by other members are removed from the set. For every member `S.M` in the set, where `S` is the type in which the member `M` is declared, the following rules are applied:
 - If `M` is a constant, field, property, event, or enumeration member, then all members declared in a base type of `S` are removed from the set.
 - If `M` is a type declaration, then all non-types declared in a base type of `S` are removed from the set, and all type declarations with the same number of type parameters as `M` declared in a base type of `S` are removed from the set.
 - If `M` is a method, then all non-method members declared in a base type of `S` are removed from the set.
- Next, interface members that are hidden by class members are removed from the set. This step only has an effect if `T` is a type parameter and `T` has both an effective base class other than `object` and a non-empty effective interface set (Type parameter constraints). For every member `S.M` in the set, where `S` is the type in which the member `M` is declared, the following rules are applied if `S` is a class declaration other than `object`:
 - If `M` is a constant, field, property, event, enumeration member, or type declaration, then all members declared in an interface declaration are removed from the set.
 - If `M` is a method, then all non-method members declared in an interface declaration are removed from the set, and all methods with the same signature as `M` declared in an interface declaration are removed from the set.

- Finally, having removed hidden members, the result of the lookup is determined:
 - If the set consists of a single member that is not a method, then this member is the result of the lookup.
 - Otherwise, if the set contains only methods, then this group of methods is the result of the lookup.
 - Otherwise, the lookup is ambiguous, and a binding-time error occurs.

For member lookups in types other than type parameters and interfaces, and member lookups in interfaces that are strictly single-inheritance (each interface in the inheritance chain has exactly zero or one direct base interface), the effect of the lookup rules is simply that derived members hide base members with the same name or signature. Such single-inheritance lookups are never ambiguous. The ambiguities that can possibly arise from member lookups in multiple-inheritance interfaces are described in [Interface member access](#).

Base types

For purposes of member lookup, a type `T` is considered to have the following base types:

- If `T` is `object`, then `T` has no base type.
- If `T` is an *enum_type*, the base types of `T` are the class types `System.Enum`, `System.ValueType`, and `object`.
- If `T` is a *struct_type*, the base types of `T` are the class types `System.ValueType` and `object`.
- If `T` is a *class_type*, the base types of `T` are the base classes of `T`, including the class type `object`.
- If `T` is an *interface_type*, the base types of `T` are the base interfaces of `T` and the class type `object`.
- If `T` is an *array_type*, the base types of `T` are the class types `System.Array` and `object`.
- If `T` is a *delegate_type*, the base types of `T` are the class types `System.Delegate` and `object`.

Function members

Function members are members that contain executable statements. Function members are always members of types and cannot be members of namespaces. C# defines the following categories of function members:

- Methods
- Properties
- Events
- Indexers
- User-defined operators
- Instance constructors
- Static constructors
- Destructors

Except for destructors and static constructors (which cannot be invoked explicitly), the statements contained in function members are executed through function member invocations. The actual syntax for writing a function member invocation depends on the particular function member category.

The argument list ([Argument lists](#)) of a function member invocation provides actual values or variable references for the parameters of the function member.

Invocations of generic methods may employ type inference to determine the set of type arguments to pass to the method. This process is described in [Type inference](#).

Invocations of methods, indexers, operators and instance constructors employ overload resolution to determine which of a candidate set of function members to invoke. This process is described in [Overload resolution](#).

Once a particular function member has been identified at binding-time, possibly through overload resolution, the actual run-time process of invoking the function member is described in [Compile-time checking of dynamic overload resolution](#).

The following table summarizes the processing that takes place in constructs involving the six categories of function members that can be explicitly invoked. In the table, `e`, `x`, `y`, and `value` indicate expressions classified as variables or values, `T` indicates an expression classified as a type, `F` is the simple name of a method, and `P` is the simple name of a property.

CONSTRUCT	EXAMPLE	DESCRIPTION
Method invocation	<code>F(x,y)</code>	Overload resolution is applied to select the best method <code>F</code> in the containing class or struct. The method is invoked with the argument list <code>(x,y)</code> . If the method is not <code>static</code> , the instance expression is <code>this</code> .
	<code>T.F(x,y)</code>	Overload resolution is applied to select the best method <code>F</code> in the class or struct <code>T</code> . A binding-time error occurs if the method is not <code>static</code> . The method is invoked with the argument list <code>(x,y)</code> .
	<code>e.F(x,y)</code>	Overload resolution is applied to select the best method <code>F</code> in the class, struct, or interface given by the type of <code>e</code> . A binding-time error occurs if the method is <code>static</code> . The method is invoked with the instance expression <code>e</code> and the argument list <code>(x,y)</code> .
Property access	<code>P</code>	The <code>get</code> accessor of the property <code>P</code> in the containing class or struct is invoked. A compile-time error occurs if <code>P</code> is write-only. If <code>P</code> is not <code>static</code> , the instance expression is <code>this</code> .
	<code>P = value</code>	The <code>set</code> accessor of the property <code>P</code> in the containing class or struct is invoked with the argument list <code>(value)</code> . A compile-time error occurs if <code>P</code> is read-only. If <code>P</code> is not <code>static</code> , the instance expression is <code>this</code> .
	<code>T.P</code>	The <code>get</code> accessor of the property <code>P</code> in the class or struct <code>T</code> is invoked. A compile-time error occurs if <code>P</code> is not <code>static</code> or if <code>P</code> is write-only.
	<code>T.P = value</code>	The <code>set</code> accessor of the property <code>P</code> in the class or struct <code>T</code> is invoked with the argument list <code>(value)</code> . A compile-time error occurs if <code>P</code> is not <code>static</code> or if <code>P</code> is read-only.

CONSTRUCT	EXAMPLE	DESCRIPTION
	<code>e.P</code>	The <code>get</code> accessor of the property <code>P</code> in the class, struct, or interface given by the type of <code>e</code> is invoked with the instance expression <code>e</code> . A binding-time error occurs if <code>P</code> is <code>static</code> or if <code>P</code> is write-only.
	<code>e.P = value</code>	The <code>set</code> accessor of the property <code>P</code> in the class, struct, or interface given by the type of <code>e</code> is invoked with the instance expression <code>e</code> and the argument list <code>(value)</code> . A binding-time error occurs if <code>P</code> is <code>static</code> or if <code>P</code> is read-only.
Event access	<code>E += value</code>	The <code>add</code> accessor of the event <code>E</code> in the containing class or struct is invoked. If <code>E</code> is not static, the instance expression is <code>this</code> .
	<code>E -= value</code>	The <code>remove</code> accessor of the event <code>E</code> in the containing class or struct is invoked. If <code>E</code> is not static, the instance expression is <code>this</code> .
	<code>T.E += value</code>	The <code>add</code> accessor of the event <code>E</code> in the class or struct <code>T</code> is invoked. A binding-time error occurs if <code>E</code> is not static.
	<code>T.E -= value</code>	The <code>remove</code> accessor of the event <code>E</code> in the class or struct <code>T</code> is invoked. A binding-time error occurs if <code>E</code> is not static.
	<code>e.E += value</code>	The <code>add</code> accessor of the event <code>E</code> in the class, struct, or interface given by the type of <code>e</code> is invoked with the instance expression <code>e</code> . A binding-time error occurs if <code>E</code> is static.
	<code>e.E -= value</code>	The <code>remove</code> accessor of the event <code>E</code> in the class, struct, or interface given by the type of <code>e</code> is invoked with the instance expression <code>e</code> . A binding-time error occurs if <code>E</code> is static.

CONSTRUCT	EXAMPLE	DESCRIPTION
Indexer access	<code>e[x,y]</code>	Overload resolution is applied to select the best indexer in the class, struct, or interface given by the type of <code>e</code> . The <code>get</code> accessor of the indexer is invoked with the instance expression <code>e</code> and the argument list <code>(x,y)</code> . A binding-time error occurs if the indexer is write-only.
	<code>e[x,y] = value</code>	Overload resolution is applied to select the best indexer in the class, struct, or interface given by the type of <code>e</code> . The <code>set</code> accessor of the indexer is invoked with the instance expression <code>e</code> and the argument list <code>(x,y,value)</code> . A binding-time error occurs if the indexer is read-only.
Operator invocation	<code>-x</code>	Overload resolution is applied to select the best unary operator in the class or struct given by the type of <code>x</code> . The selected operator is invoked with the argument list <code>(x)</code> .
	<code>x + y</code>	Overload resolution is applied to select the best binary operator in the classes or structs given by the types of <code>x</code> and <code>y</code> . The selected operator is invoked with the argument list <code>(x,y)</code> .
Instance constructor invocation	<code>new T(x,y)</code>	Overload resolution is applied to select the best instance constructor in the class or struct <code>T</code> . The instance constructor is invoked with the argument list <code>(x,y)</code> .

Argument lists

Every function member and delegate invocation includes an argument list which provides actual values or variable references for the parameters of the function member. The syntax for specifying the argument list of a function member invocation depends on the function member category:

- For instance constructors, methods, indexers and delegates, the arguments are specified as an *argument_list*, as described below. For indexers, when invoking the `set` accessor, the argument list additionally includes the expression specified as the right operand of the assignment operator.
- For properties, the argument list is empty when invoking the `get` accessor, and consists of the expression specified as the right operand of the assignment operator when invoking the `set` accessor.
- For events, the argument list consists of the expression specified as the right operand of the `+=` or `-=` operator.
- For user-defined operators, the argument list consists of the single operand of the unary operator or the two operands of the binary operator.

The arguments of properties ([Properties](#)), events ([Events](#)), and user-defined operators ([Operators](#)) are always passed as value parameters ([Value parameters](#)). The arguments of indexers ([Indexers](#)) are always passed as

value parameters ([Value parameters](#)) or parameter arrays ([Parameter arrays](#)). Reference and output parameters are not supported for these categories of function members.

The arguments of an instance constructor, method, indexer or delegate invocation are specified as an *argument_list*.

```
argument_list
    : argument (',' argument)*
    ;

argument
    : argument_name? argument_value
    ;

argument_name
    : identifier ':'
    ;

argument_value
    : expression
    | 'ref' variable_reference
    | 'out' variable_reference
    ;
```

An *argument_list* consists of one or more *arguments*, separated by commas. Each argument consists of an optional *argument_name* followed by an *argument_value*. An *argument* with an *argument_name* is referred to as a **named argument**, whereas an *argument* without an *argument_name* is a **positional argument**. It is an error for a positional argument to appear after a named argument in an *argument_list*.

The *argument_value* can take one of the following forms:

- An *expression*, indicating that the argument is passed as a value parameter ([Value parameters](#)).
- The keyword `ref` followed by a *variable_reference* ([Variable references](#)), indicating that the argument is passed as a reference parameter ([Reference parameters](#)). A variable must be definitely assigned ([Definite assignment](#)) before it can be passed as a reference parameter. The keyword `out` followed by a *variable_reference* ([Variable references](#)), indicating that the argument is passed as an output parameter ([Output parameters](#)). A variable is considered definitely assigned ([Definite assignment](#)) following a function member invocation in which the variable is passed as an output parameter.

Corresponding parameters

For each argument in an argument list there has to be a corresponding parameter in the function member or delegate being invoked.

The parameter list used in the following is determined as follows:

- For virtual methods and indexers defined in classes, the parameter list is picked from the most specific declaration or override of the function member, starting with the static type of the receiver, and searching through its base classes.
- For interface methods and indexers, the parameter list is picked from the most specific definition of the member, starting with the interface type and searching through the base interfaces. If no unique parameter list is found, a parameter list with inaccessible names and no optional parameters is constructed, so that invocations cannot use named parameters or omit optional arguments.
- For partial methods, the parameter list of the defining partial method declaration is used.
- For all other function members and delegates there is only a single parameter list, which is the one used.

The position of an argument or parameter is defined as the number of arguments or parameters preceding it in the argument list or parameter list.

The corresponding parameters for function member arguments are established as follows:

- Arguments in the *argument_list* of instance constructors, methods, indexers and delegates:
 - A positional argument where a fixed parameter occurs at the same position in the parameter list corresponds to that parameter.
 - A positional argument of a function member with a parameter array invoked in its normal form corresponds to the parameter array, which must occur at the same position in the parameter list.
 - A positional argument of a function member with a parameter array invoked in its expanded form, where no fixed parameter occurs at the same position in the parameter list, corresponds to an element in the parameter array.
 - A named argument corresponds to the parameter of the same name in the parameter list.
 - For indexers, when invoking the `set` accessor, the expression specified as the right operand of the assignment operator corresponds to the implicit `value` parameter of the `set` accessor declaration.
- For properties, when invoking the `get` accessor there are no arguments. When invoking the `set` accessor, the expression specified as the right operand of the assignment operator corresponds to the implicit `value` parameter of the `set` accessor declaration.
- For user-defined unary operators (including conversions), the single operand corresponds to the single parameter of the operator declaration.
- For user-defined binary operators, the left operand corresponds to the first parameter, and the right operand corresponds to the second parameter of the operator declaration.

Run-time evaluation of argument lists

During the run-time processing of a function member invocation ([Compile-time checking of dynamic overload resolution](#)), the expressions or variable references of an argument list are evaluated in order, from left to right, as follows:

- For a value parameter, the argument expression is evaluated and an implicit conversion ([Implicit conversions](#)) to the corresponding parameter type is performed. The resulting value becomes the initial value of the value parameter in the function member invocation.
- For a reference or output parameter, the variable reference is evaluated and the resulting storage location becomes the storage location represented by the parameter in the function member invocation. If the variable reference given as a reference or output parameter is an array element of a *reference_type*, a run-time check is performed to ensure that the element type of the array is identical to the type of the parameter. If this check fails, a `System.ArrayTypeMismatchException` is thrown.

Methods, indexers, and instance constructors may declare their right-most parameter to be a parameter array ([Parameter arrays](#)). Such function members are invoked either in their normal form or in their expanded form depending on which is applicable ([Applicable function member](#)):

- When a function member with a parameter array is invoked in its normal form, the argument given for the parameter array must be a single expression that is implicitly convertible ([Implicit conversions](#)) to the parameter array type. In this case, the parameter array acts precisely like a value parameter.
- When a function member with a parameter array is invoked in its expanded form, the invocation must specify zero or more positional arguments for the parameter array, where each argument is an expression that is implicitly convertible ([Implicit conversions](#)) to the element type of the parameter array. In this case, the invocation creates an instance of the parameter array type with a length corresponding to the number of arguments, initializes the elements of the array instance with the given argument values, and uses the newly created array instance as the actual argument.

The expressions of an argument list are always evaluated in the order they are written. Thus, the example


```

class Test
{
    static void F(int x, int y = -1, int z = -2) {
        System.Console.WriteLine("x = {0}, y = {1}, z = {2}", x, y, z);
    }

    static void Main() {
        int i = 0;
        F(i++, i++, i++);
        F(z: i++, x: i++);
    }
}

```

produces the output

```

x = 0, y = 1, z = 2
x = 4, y = -1, z = 3

```

The array co-variance rules ([Array covariance](#)) permit a value of an array type `A[]` to be a reference to an instance of an array type `B[]`, provided an implicit reference conversion exists from `B` to `A`. Because of these rules, when an array element of a *reference type* is passed as a reference or output parameter, a run-time check is required to ensure that the actual element type of the array is identical to that of the parameter. In the example

```

class Test
{
    static void F(ref object x) {...}

    static void Main() {
        object[] a = new object[10];
        object[] b = new string[10];
        F(ref a[0]);           // Ok
        F(ref b[1]);           // ArrayTypeMismatchException
    }
}

```

the second invocation of `F` causes a `System.ArrayTypeMismatchException` to be thrown because the actual element type of `b` is `string` and not `object`.

When a function member with a parameter array is invoked in its expanded form, the invocation is processed exactly as if an array creation expression with an array initializer ([Array creation expressions](#)) was inserted around the expanded parameters. For example, given the declaration

```

void F(int x, int y, params object[] args);

```

the following invocations of the expanded form of the method

```

F(10, 20);
F(10, 20, 30, 40);
F(10, 20, 1, "hello", 3.0);

```

correspond exactly to

```

F(10, 20, new object[] {});
F(10, 20, new object[] {30, 40});
F(10, 20, new object[] {1, "hello", 3.0});

```

In particular, note that an empty array is created when there are zero arguments given for the parameter array.

When arguments are omitted from a function member with corresponding optional parameters, the default arguments of the function member declaration are implicitly passed. Because these are always constant, their evaluation will not impact the evaluation order of the remaining arguments.

Type inference

When a generic method is called without specifying type arguments, a *type inference* process attempts to infer type arguments for the call. The presence of type inference allows a more convenient syntax to be used for calling a generic method, and allows the programmer to avoid specifying redundant type information. For example, given the method declaration:

```
class Chooser
{
    static Random rand = new Random();

    public static T Choose<T>(T first, T second) {
        return (rand.Next(2) == 0)? first: second;
    }
}
```

it is possible to invoke the `Choose` method without explicitly specifying a type argument:

```
int i = Chooser.Choose(5, 213);           // Calls Choose<int>

string s = Chooser.Choose("foo", "bar");   // Calls Choose<string>
```

Through type inference, the type arguments `int` and `string` are determined from the arguments to the method.

Type inference occurs as part of the binding-time processing of a method invocation ([Method invocations](#)) and takes place before the overload resolution step of the invocation. When a particular method group is specified in a method invocation, and no type arguments are specified as part of the method invocation, type inference is applied to each generic method in the method group. If type inference succeeds, then the inferred type arguments are used to determine the types of arguments for subsequent overload resolution. If overload resolution chooses a generic method as the one to invoke, then the inferred type arguments are used as the actual type arguments for the invocation. If type inference for a particular method fails, that method does not participate in overload resolution. The failure of type inference, in and of itself, does not cause a binding-time error. However, it often leads to a binding-time error when overload resolution then fails to find any applicable methods.

If the supplied number of arguments is different than the number of parameters in the method, then inference immediately fails. Otherwise, assume that the generic method has the following signature:

```
Tr M<X1,...,Xn>(T1 x1, ..., Tm xm)
```

With a method call of the form `M(E1...Em)` the task of type inference is to find unique type arguments `S1...Sn` for each of the type parameters `X1...Xn` so that the call `M<S1...Sn>(E1...Em)` becomes valid.

During the process of inference each type parameter `xi` is either *fixed* to a particular type `si` or *unfixed* with an associated set of *bounds*. Each of the bounds is some type `T`. Initially each type variable `xi` is unfixed with an empty set of bounds.

Type inference takes place in phases. Each phase will try to infer type arguments for more type variables based on the findings of the previous phase. The first phase makes some initial inferences of bounds, whereas the

second phase fixes type variables to specific types and infers further bounds. The second phase may have to be repeated a number of times.

Note: Type inference takes place not only when a generic method is called. Type inference for conversion of method groups is described in [Type inference for conversion of method groups](#) and finding the best common type of a set of expressions is described in [Finding the best common type of a set of expressions](#).

The first phase

For each of the method arguments E_i :

- If E_i is an anonymous function, an *explicit parameter type inference* ([Explicit parameter type inferences](#)) is made from E_i to T_i .
- Otherwise, if E_i has a type U and x_i is a value parameter then a *lower-bound inference* is made from U to T_i .
- Otherwise, if E_i has a type U and x_i is a `ref` or `out` parameter then an *exact inference* is made from U to T_i .
- Otherwise, no inference is made for this argument.

The second phase

The second phase proceeds as follows:

- All *unfixed* type variables x_i which do not *depend on* ([Dependence](#)) any x_j are fixed ([Fixing](#)).
- If no such type variables exist, all *unfixed* type variables x_i are *fixed* for which all of the following hold:
 - There is at least one type variable x_j that depends on x_i
 - x_i has a non-empty set of bounds
- If no such type variables exist and there are still *unfixed* type variables, type inference fails.
- Otherwise, if no further *unfixed* type variables exist, type inference succeeds.
- Otherwise, for all arguments E_i with corresponding parameter type T_i where the *output types* ([Output types](#)) contain *unfixed* type variables x_j but the *input types* ([Input types](#)) do not, an *output type inference* ([Output type inferences](#)) is made from E_i to T_i . Then the second phase is repeated.

Input types

If E is a method group or implicitly typed anonymous function and T is a delegate type or expression tree type then all the parameter types of T are *input types* of E with type T .

Output types

If E is a method group or an anonymous function and T is a delegate type or expression tree type then the return type of T is an *output type* of E with type T .

Dependence

An *unfixed* type variable x_i *depends directly on* an unfixed type variable x_j if for some argument E_k with type T_k x_j occurs in an *input type* of E_k with type T_k and x_i occurs in an *output type* of E_k with type T_k .

x_j *depends on* x_i if x_j *depends directly on* x_i or if x_i *depends directly on* x_k and x_k *depends on* x_j . Thus "depends on" is the transitive but not reflexive closure of "depends directly on".

Output type inferences

An *output type inference* is made from an expression E to a type T in the following way:

- If E is an anonymous function with inferred return type U ([Inferred return type](#)) and T is a delegate type or expression tree type with return type T_b , then a *lower-bound inference* ([Lower-bound inferences](#)) is made from U to T_b .
- Otherwise, if E is a method group and T is a delegate type or expression tree type with parameter types $T_1 \dots T_k$ and return type T_b , and overload resolution of E with the types $T_1 \dots T_k$ yields a single method

with return type U , then a *lower-bound inference* is made from U to T_b .

- Otherwise, if E is an expression with type U , then a *lower-bound inference* is made from U to T .
- Otherwise, no inferences are made.

Explicit parameter type inferences

An *explicit parameter type inference* is made from an expression E to a type T in the following way:

- If E is an explicitly typed anonymous function with parameter types $U_1 \dots U_k$ and T is a delegate type or expression tree type with parameter types $V_1 \dots V_k$ then for each U_i an *exact inference* ([Exact inferences](#)) is made from U_i to the corresponding V_i .

Exact inferences

An *exact inference* from a type U to a type V is made as follows:

- If V is one of the *unfixed* X_i then U is added to the set of exact bounds for X_i .
- Otherwise, sets $V_1 \dots V_k$ and $U_1 \dots U_k$ are determined by checking if any of the following cases apply:
 - V is an array type $V_1[\dots]$ and U is an array type $U_1[\dots]$ of the same rank
 - V is the type $V_1?$ and U is the type $U_1?$
 - V is a constructed type $C\langle V_1 \dots V_k \rangle$ and U is a constructed type $C\langle U_1 \dots U_k \rangle$

If any of these cases apply then an *exact inference* is made from each U_i to the corresponding V_i .

- Otherwise no inferences are made.

Lower-bound inferences

A *lower-bound inference* from a type U to a type V is made as follows:

- If V is one of the *unfixed* X_i then U is added to the set of lower bounds for X_i .
- Otherwise, if V is the type $V_1?$ and U is the type $U_1?$ then a lower bound inference is made from U_1 to V_1 .
- Otherwise, sets $U_1 \dots U_k$ and $V_1 \dots V_k$ are determined by checking if any of the following cases apply:
 - V is an array type $V_1[\dots]$ and U is an array type $U_1[\dots]$ (or a type parameter whose effective base type is $U_1[\dots]$) of the same rank
 - V is one of `IEnumerable<V1>`, `ICollection<V1>` or `IList<V1>` and U is a one-dimensional array type $U_1[]$ (or a type parameter whose effective base type is $U_1[]$)
 - V is a constructed class, struct, interface or delegate type $C\langle V_1 \dots V_k \rangle$ and there is a unique type $C\langle U_1 \dots U_k \rangle$ such that U (or, if U is a type parameter, its effective base class or any member of its effective interface set) is identical to, inherits from (directly or indirectly), or implements (directly or indirectly) $C\langle U_1 \dots U_k \rangle$.

(The "uniqueness" restriction means that in the case interface `C<T> {} class U: C<X>, C<Y> {}`, then no inference is made when inferring from U to $C\langle T \rangle$ because U_1 could be X or Y .)

If any of these cases apply then an inference is made from each U_i to the corresponding V_i as follows:

- If U_i is not known to be a reference type then an *exact inference* is made
- Otherwise, if U is an array type then a *lower-bound inference* is made
- Otherwise, if V is $C\langle V_1 \dots V_k \rangle$ then inference depends on the i -th type parameter of C :
 - If it is covariant then a *lower-bound inference* is made.
 - If it is contravariant then an *upper-bound inference* is made.
 - If it is invariant then an *exact inference* is made.

- Otherwise, no inferences are made.

Upper-bound inferences

An *upper-bound inference* from a type u to a type v is made as follows:

- If v is one of the *unfixed* x_i then u is added to the set of upper bounds for x_i .
- Otherwise, sets $v_1 \dots v_k$ and $u_1 \dots u_k$ are determined by checking if any of the following cases apply:
 - u is an array type $u_1[\dots]$ and v is an array type $v_1[\dots]$ of the same rank
 - u is one of `IEnumerable<Ue>`, `ICollection<Ue>` or `IList<Ue>` and v is a one-dimensional array type $ve[]$
 - u is the type $u_1?$ and v is the type $v_1?$
 - u is constructed class, struct, interface or delegate type $c<u_1 \dots u_k>$ and v is a class, struct, interface or delegate type which is identical to, inherits from (directly or indirectly), or implements (directly or indirectly) a unique type $c<v_1 \dots v_k>$

(The "uniqueness" restriction means that if we have

`interface C<T>{} class V<Z>: C<X<Z>>, C<Y<Z>>{} , then no inference is made when inferring from $c<u_1>$ to $v<Q>$. Inferences are not made from u_1 to either $x<Q>$ or $y<Q>$.)`

If any of these cases apply then an inference is made *from* each u_i *to* the corresponding v_i as follows:

- If u_i is not known to be a reference type then an *exact inference* is made
- Otherwise, if v is an array type then an *upper-bound inference* is made
- Otherwise, if u is $c<u_1 \dots u_k>$ then inference depends on the i -th type parameter of c :
 - If it is covariant then an *upper-bound inference* is made.
 - If it is contravariant then a *lower-bound inference* is made.
 - If it is invariant then an *exact inference* is made.
- Otherwise, no inferences are made.

Fixing

An *unfixed* type variable x_i with a set of bounds is *fixed* as follows:

- The set of *candidate types* u_j starts out as the set of all types in the set of bounds for x_i .
- We then examine each bound for x_i in turn: For each exact bound u of x_i all types u_j which are not identical to u are removed from the candidate set. For each lower bound u of x_i all types u_j to which there is *not* an implicit conversion from u are removed from the candidate set. For each upper bound u of x_i all types u_j from which there is *not* an implicit conversion to u are removed from the candidate set.
- If among the remaining candidate types u_j there is a unique type v from which there is an implicit conversion to all the other candidate types, then x_i is fixed to v .
- Otherwise, type inference fails.

Inferred return type

The inferred return type of an anonymous function F is used during type inference and overload resolution. The inferred return type can only be determined for an anonymous function where all parameter types are known, either because they are explicitly given, provided through an anonymous function conversion or inferred during type inference on an enclosing generic method invocation.

The *inferred result type* is determined as follows:

- If the body of F is an *expression* that has a type, then the inferred result type of F is the type of that expression.

- If the body of `F` is a *block* and the set of expressions in the block's `return` statements has a best common type `T` ([Finding the best common type of a set of expressions](#)), then the inferred result type of `F` is `T`.
- Otherwise, a result type cannot be inferred for `F`.

The *inferred return type* is determined as follows:

- If `F` is `async` and the body of `F` is either an expression classified as nothing ([Expression classifications](#)), or a statement block where no return statements have expressions, the inferred return type is `System.Threading.Tasks.Task`.
- If `F` is `async` and has an inferred result type `T`, the inferred return type is `System.Threading.Tasks.Task<T>`.
- If `F` is non-`async` and has an inferred result type `T`, the inferred return type is `T`.
- Otherwise a return type cannot be inferred for `F`.

As an example of type inference involving anonymous functions, consider the `Select` extension method declared in the `System.Linq.Enumerable` class:

```
namespace System.Linq
{
    public static class Enumerable
    {
        public static IEnumerable<TResult> Select<TSource,TResult>(
            this IEnumerable<TSource> source,
            Func<TSource,TResult> selector)
        {
            foreach (TSource element in source) yield return selector(element);
        }
    }
}
```

Assuming the `System.Linq` namespace was imported with a `using` clause, and given a class `Customer` with a `Name` property of type `string`, the `Select` method can be used to select the names of a list of customers:

```
List<Customer> customers = GetCustomerList();
IEnumerable<string> names = customers.Select(c => c.Name);
```

The extension method invocation ([Extension method invocations](#)) of `Select` is processed by rewriting the invocation to a static method invocation:

```
IEnumerable<string> names = Enumerable.Select(customers, c => c.Name);
```

Since type arguments were not explicitly specified, type inference is used to infer the type arguments. First, the `customers` argument is related to the `source` parameter, inferring `T` to be `Customer`. Then, using the anonymous function type inference process described above, `c` is given type `Customer`, and the expression `c.Name` is related to the return type of the `selector` parameter, inferring `S` to be `string`. Thus, the invocation is equivalent to

```
Sequence.Select<Customer,string>(customers, (Customer c) => c.Name)
```

and the result is of type `IEnumerable<string>`.

The following example demonstrates how anonymous function type inference allows type information to "flow" between arguments in a generic method invocation. Given the method:

```
static Z F<X,Y,Z>(X value, Func<X,Y> f1, Func<Y,Z> f2) {
    return f2(f1(value));
}
```

Type inference for the invocation:

```
double seconds = F("1:15:30", s => TimeSpan.Parse(s), t => t.TotalSeconds);
```

proceeds as follows: First, the argument `"1:15:30"` is related to the `value` parameter, inferring `X` to be `string`. Then, the parameter of the first anonymous function, `s`, is given the inferred type `string`, and the expression `TimeSpan.Parse(s)` is related to the return type of `f1`, inferring `Y` to be `System.TimeSpan`. Finally, the parameter of the second anonymous function, `t`, is given the inferred type `System.TimeSpan`, and the expression `t.TotalSeconds` is related to the return type of `f2`, inferring `Z` to be `double`. Thus, the result of the invocation is of type `double`.

Type inference for conversion of method groups

Similar to calls of generic methods, type inference must also be applied when a method group `M` containing a generic method is converted to a given delegate type `D` ([Method group conversions](#)). Given a method

```
Tr M<X1...Xn>(T1 x1 ... Tm xm)
```

and the method group `M` being assigned to the delegate type `D` the task of type inference is to find type arguments `S1...Sn` so that the expression:

```
M<S1...Sn>
```

becomes compatible ([Delegate declarations](#)) with `D`.

Unlike the type inference algorithm for generic method calls, in this case there are only argument *types*, no argument *expressions*. In particular, there are no anonymous functions and hence no need for multiple phases of inference.

Instead, all `xi` are considered *unfixed*, and a *lower-bound inference* is made *from* each argument type `uj` of `D` to the corresponding parameter type `Tj` of `M`. If for any of the `xi` no bounds were found, type inference fails. Otherwise, all `xi` are *fixed* to corresponding `si`, which are the result of type inference.

Finding the best common type of a set of expressions

In some cases, a common type needs to be inferred for a set of expressions. In particular, the element types of implicitly typed arrays and the return types of anonymous functions with *block* bodies are found in this way.

Intuitively, given a set of expressions `E1...Em` this inference should be equivalent to calling a method

```
Tr M<X>(X x1 ... X xm)
```

with the `Ei` as arguments.

More precisely, the inference starts out with an *unfixed* type variable `x`. *Output type inferences* are then made *from* each `Ei` to `x`. Finally, `x` is *fixed* and, if successful, the resulting type `s` is the resulting best common type for the expressions. If no such `s` exists, the expressions have no best common type.

Overload resolution

Overload resolution is a binding-time mechanism for selecting the best function member to invoke given an

argument list and a set of candidate function members. Overload resolution selects the function member to invoke in the following distinct contexts within C#:

- Invocation of a method named in an *invocation_expression* ([Method invocations](#)).
- Invocation of an instance constructor named in an *object_creation_expression* ([Object creation expressions](#)).
- Invocation of an indexer accessor through an *element_access* ([Element access](#)).
- Invocation of a predefined or user-defined operator referenced in an expression ([Unary operator overload resolution](#) and [Binary operator overload resolution](#)).

Each of these contexts defines the set of candidate function members and the list of arguments in its own unique way, as described in detail in the sections listed above. For example, the set of candidates for a method invocation does not include methods marked `override` ([Member lookup](#)), and methods in a base class are not candidates if any method in a derived class is applicable ([Method invocations](#)).

Once the candidate function members and the argument list have been identified, the selection of the best function member is the same in all cases:

- Given the set of applicable candidate function members, the best function member in that set is located. If the set contains only one function member, then that function member is the best function member. Otherwise, the best function member is the one function member that is better than all other function members with respect to the given argument list, provided that each function member is compared to all other function members using the rules in [Better function member](#). If there is not exactly one function member that is better than all other function members, then the function member invocation is ambiguous and a binding-time error occurs.

The following sections define the exact meanings of the terms *applicable function member* and *better function member*.

Applicable function member

A function member is said to be an *applicable function member* with respect to an argument list `A` when all of the following are true:

- Each argument in `A` corresponds to a parameter in the function member declaration as described in [Corresponding parameters](#), and any parameter to which no argument corresponds is an optional parameter.
- For each argument in `A`, the parameter passing mode of the argument (i.e., value, `ref`, or `out`) is identical to the parameter passing mode of the corresponding parameter, and
 - for a value parameter or a parameter array, an implicit conversion ([Implicit conversions](#)) exists from the argument to the type of the corresponding parameter, or
 - for a `ref` or `out` parameter, the type of the argument is identical to the type of the corresponding parameter. After all, a `ref` or `out` parameter is an alias for the argument passed.

For a function member that includes a parameter array, if the function member is applicable by the above rules, it is said to be applicable in its *normal form*. If a function member that includes a parameter array is not applicable in its normal form, the function member may instead be applicable in its *expanded form*:

- The expanded form is constructed by replacing the parameter array in the function member declaration with zero or more value parameters of the element type of the parameter array such that the number of arguments in the argument list `A` matches the total number of parameters. If `A` has fewer arguments than the number of fixed parameters in the function member declaration, the expanded form of the function member cannot be constructed and is thus not applicable.
- Otherwise, the expanded form is applicable if for each argument in `A` the parameter passing mode of the argument is identical to the parameter passing mode of the corresponding parameter, and
 - for a fixed value parameter or a value parameter created by the expansion, an implicit conversion ([Implicit conversions](#)) exists from the type of the argument to the type of the corresponding parameter, or

- for a `ref` or `out` parameter, the type of the argument is identical to the type of the corresponding parameter.

Better function member

For the purposes of determining the better function member, a stripped-down argument list A is constructed containing just the argument expressions themselves in the order they appear in the original argument list.

Parameter lists for each of the candidate function members are constructed in the following way:

- The expanded form is used if the function member was applicable only in the expanded form.
- Optional parameters with no corresponding arguments are removed from the parameter list
- The parameters are reordered so that they occur at the same position as the corresponding argument in the argument list.

Given an argument list A with a set of argument expressions $\{E_1, E_2, \dots, E_n\}$ and two applicable function members M_p and M_q with parameter types $\{P_1, P_2, \dots, P_n\}$ and $\{Q_1, Q_2, \dots, Q_n\}$, M_p is defined to be a **better function member** than M_q if

- for each argument, the implicit conversion from E_x to Q_x is not better than the implicit conversion from E_x to P_x , and
- for at least one argument, the conversion from E_x to P_x is better than the conversion from E_x to Q_x .

When performing this evaluation, if M_p or M_q is applicable in its expanded form, then P_x or Q_x refers to a parameter in the expanded form of the parameter list.

In case the parameter type sequences $\{P_1, P_2, \dots, P_n\}$ and $\{Q_1, Q_2, \dots, Q_n\}$ are equivalent (i.e. each P_i has an identity conversion to the corresponding Q_i), the following tie-breaking rules are applied, in order, to determine the better function member.

- If M_p is a non-generic method and M_q is a generic method, then M_p is better than M_q .
- Otherwise, if M_p is applicable in its normal form and M_q has a `params` array and is applicable only in its expanded form, then M_p is better than M_q .
- Otherwise, if M_p has more declared parameters than M_q , then M_p is better than M_q . This can occur if both methods have `params` arrays and are applicable only in their expanded forms.
- Otherwise if all parameters of M_p have a corresponding argument whereas default arguments need to be substituted for at least one optional parameter in M_q then M_p is better than M_q .
- Otherwise, if M_p has more specific parameter types than M_q , then M_p is better than M_q . Let $\{R_1, R_2, \dots, R_n\}$ and $\{S_1, S_2, \dots, S_n\}$ represent the uninstantiated and unexpanded parameter types of M_p and M_q . M_p 's parameter types are more specific than M_q 's if, for each parameter, R_x is not less specific than S_x , and, for at least one parameter, R_x is more specific than S_x :
 - A type parameter is less specific than a non-type parameter.
 - Recursively, a constructed type is more specific than another constructed type (with the same number of type arguments) if at least one type argument is more specific and no type argument is less specific than the corresponding type argument in the other.
 - An array type is more specific than another array type (with the same number of dimensions) if the element type of the first is more specific than the element type of the second.
- Otherwise if one member is a non-lifted operator and the other is a lifted operator, the non-lifted one is better.
- Otherwise, neither function member is better.

Better conversion from expression

Given an implicit conversion C_1 that converts from an expression E to a type T_1 , and an implicit conversion C_2 that converts from an expression E to a type T_2 , C_1 is a **better conversion** than C_2 if E does not

exactly match `T2` and at least one of the following holds:

- `E` exactly matches `T1` ([Exactly matching Expression](#))
- `T1` is a better conversion target than `T2` ([Better conversion target](#))

Exactly matching Expression

Given an expression `E` and a type `T`, `E` exactly matches `T` if one of the following holds:

- `E` has a type `S`, and an identity conversion exists from `S` to `T`
- `E` is an anonymous function, `T` is either a delegate type `D` or an expression tree type `Expression<D>` and one of the following holds:
 - An inferred return type `X` exists for `E` in the context of the parameter list of `D` ([Inferred return type](#)), and an identity conversion exists from `X` to the return type of `D`
 - Either `E` is non-async and `D` has a return type `Y` or `E` is async and `D` has a return type `Task<Y>`, and one of the following holds:
 - The body of `E` is an expression that exactly matches `Y`
 - The body of `E` is a statement block where every return statement returns an expression that exactly matches `Y`

Better conversion target

Given two different types `T1` and `T2`, `T1` is a better conversion target than `T2` if no implicit conversion from `T2` to `T1` exists, and at least one of the following holds:

- An implicit conversion from `T1` to `T2` exists
- `T1` is either a delegate type `D1` or an expression tree type `Expression<D1>`, `T2` is either a delegate type `D2` or an expression tree type `Expression<D2>`, `D1` has a return type `S1` and one of the following holds:
 - `D2` is void returning
 - `D2` has a return type `S2`, and `S1` is a better conversion target than `S2`
- `T1` is `Task<S1>`, `T2` is `Task<S2>`, and `S1` is a better conversion target than `S2`
- `T1` is `S1` or `S1?` where `S1` is a signed integral type, and `T2` is `S2` or `S2?` where `S2` is an unsigned integral type. Specifically:
 - `S1` is `sbyte` and `S2` is `byte`, `ushort`, `uint`, or `ulong`
 - `S1` is `short` and `S2` is `ushort`, `uint`, or `ulong`
 - `S1` is `int` and `S2` is `uint`, or `ulong`
 - `S1` is `long` and `S2` is `ulong`

Overloading in generic classes

While signatures as declared must be unique, it is possible that substitution of type arguments results in identical signatures. In such cases, the tie-breaking rules of overload resolution above will pick the most specific member.

The following examples show overloads that are valid and invalid according to this rule:

```

interface I1<T> {...}

interface I2<T> {...}

class G1<U>
{
    int F1(U u);           // Overload resolution for G<int>.F1
    int F1(int i);         // will pick non-generic

    void F2(I1<U> a);       // Valid overload
    void F2(I2<U> a);
}

class G2<U,V>
{
    void F3(U u, V v);      // Valid, but overload resolution for
    void F3(V v, U u);      // G2<int,int>.F3 will fail

    void F4(U u, I1<V> v);  // Valid, but overload resolution for
    void F4(I1<V> v, U u);  // G2<I1<int>,int>.F4 will fail

    void F5(U u1, I1<V> v2); // Valid overload
    void F5(V v1, U u2);

    void F6(ref U u);        // valid overload
    void F6(out V v);
}

```

Compile-time checking of dynamic overload resolution

For most dynamically bound operations the set of possible candidates for resolution is unknown at compile-time. In certain cases, however the candidate set is known at compile-time:

- Static method calls with dynamic arguments
- Instance method calls where the receiver is not a dynamic expression
- Indexer calls where the receiver is not a dynamic expression
- Constructor calls with dynamic arguments

In these cases a limited compile-time check is performed for each candidate to see if any of them could possibly apply at run-time. This check consists of the following steps:

- Partial type inference: Any type argument that does not depend directly or indirectly on an argument of type `dynamic` is inferred using the rules of [Type inference](#). The remaining type arguments are unknown.
- Partial applicability check: Applicability is checked according to [Applicable function member](#), but ignoring parameters whose types are unknown.
- If no candidate passes this test, a compile-time error occurs.

Function member invocation

This section describes the process that takes place at run-time to invoke a particular function member. It is assumed that a binding-time process has already determined the particular member to invoke, possibly by applying overload resolution to a set of candidate function members.

For purposes of describing the invocation process, function members are divided into two categories:

- Static function members. These are instance constructors, static methods, static property accessors, and user-defined operators. Static function members are always non-virtual.
- Instance function members. These are instance methods, instance property accessors, and indexer accessors. Instance function members are either non-virtual or virtual, and are always invoked on a particular instance. The instance is computed by an instance expression, and it becomes accessible within the function member as `this` ([This access](#)).

The run-time processing of a function member invocation consists of the following steps, where `M` is the function member and, if `M` is an instance member, `E` is the instance expression:

- If `M` is a static function member:
 - The argument list is evaluated as described in [Argument lists](#).
 - `M` is invoked.
- If `M` is an instance function member declared in a *value_type*:
 - `E` is evaluated. If this evaluation causes an exception, then no further steps are executed.
 - If `E` is not classified as a variable, then a temporary local variable of `E`'s type is created and the value of `E` is assigned to that variable. `E` is then reclassified as a reference to that temporary local variable. The temporary variable is accessible as `this` within `M`, but not in any other way. Thus, only when `E` is a true variable is it possible for the caller to observe the changes that `M` makes to `this`.
 - The argument list is evaluated as described in [Argument lists](#).
 - `M` is invoked. The variable referenced by `E` becomes the variable referenced by `this`.
- If `M` is an instance function member declared in a *reference_type*:
 - `E` is evaluated. If this evaluation causes an exception, then no further steps are executed.
 - The argument list is evaluated as described in [Argument lists](#).
 - If the type of `E` is a *value_type*, a boxing conversion ([Boxing conversions](#)) is performed to convert `E` to type `object`, and `E` is considered to be of type `object` in the following steps. In this case, `M` could only be a member of `System.Object`.
 - The value of `E` is checked to be valid. If the value of `E` is `null`, a `System.NullReferenceException` is thrown and no further steps are executed.
 - The function member implementation to invoke is determined:
 - If the binding-time type of `E` is an interface, the function member to invoke is the implementation of `M` provided by the run-time type of the instance referenced by `E`. This function member is determined by applying the interface mapping rules ([Interface mapping](#)) to determine the implementation of `M` provided by the run-time type of the instance referenced by `E`.
 - Otherwise, if `M` is a virtual function member, the function member to invoke is the implementation of `M` provided by the run-time type of the instance referenced by `E`. This function member is determined by applying the rules for determining the most derived implementation ([Virtual methods](#)) of `M` with respect to the run-time type of the instance referenced by `E`.
 - Otherwise, `M` is a non-virtual function member, and the function member to invoke is `M` itself.
 - The function member implementation determined in the step above is invoked. The object referenced by `E` becomes the object referenced by `this`.

Invocations on boxed instances

A function member implemented in a *value_type* can be invoked through a boxed instance of that *value_type* in the following situations:

- When the function member is an `override` of a method inherited from type `object` and is invoked through an instance expression of type `object`.
- When the function member is an implementation of an interface function member and is invoked through an instance expression of an *interface_type*.
- When the function member is invoked through a delegate.

In these situations, the boxed instance is considered to contain a variable of the *value_type*, and this variable becomes the variable referenced by `this` within the function member invocation. In particular, this means that

when a function member is invoked on a boxed instance, it is possible for the function member to modify the value contained in the boxed instance.

Primary expressions

Primary expressions include the simplest forms of expressions.

```
primary_expression
: primary_no_array_creation_expression
| array_creation_expression
;

primary_no_array_creation_expression
: literal
| interpolated_string_expression
| simple_name
| parenthesized_expression
| member_access
| invocation_expression
| element_access
| this_access
| base_access
| post_increment_expression
| post_decrement_expression
| object_creation_expression
| delegate_creation_expression
| anonymous_object_creation_expression
| typeof_expression
| checked_expression
| unchecked_expression
| default_value_expression
| nameof_expression
| anonymous_method_expression
| primary_no_array_creation_expression_unsafe
;
```

Primary expressions are divided between *array_creation_expressions* and *primary_no_array_creation_expressions*. Treating array-creation-expression in this way, rather than listing it along with the other simple expression forms, enables the grammar to disallow potentially confusing code such as

```
object o = new int[3][1];
```

which would otherwise be interpreted as

```
object o = (new int[3])[1];
```

Literals

A *primary_expression* that consists of a *literal* ([Literals](#)) is classified as a value.

Interpolated strings

An *interpolated_string_expression* consists of a `$` sign followed by a regular or verbatim string literal, wherein holes, delimited by `{` and `}`, enclose expressions and formatting specifications. An interpolated string expression is the result of an *interpolated_string_literal* that has been broken up into individual tokens, as described in [Interpolated string literals](#).

```

interpolated_string_expression
    : '$' interpolated_regular_string
    | '$' interpolated_verbatim_string
    ;

interpolated_regular_string
    : interpolated_regular_string_whole
    | interpolated_regular_string_start interpolated_regular_string_body interpolated_regular_string_end
    ;

interpolated_regular_string_body
    : interpolation (interpolated_regular_string_mid interpolation)*
    ;

interpolation
    : expression
    | expression ',' constant_expression
    ;

interpolated_verbatim_string
    : interpolated_verbatim_string_whole
    | interpolated_verbatim_string_start interpolated_verbatim_string_body interpolated_verbatim_string_end
    ;

interpolated_verbatim_string_body
    : interpolation (interpolated_verbatim_string_mid interpolation)+
    ;

```

The *constant_expression* in an interpolation must have an implicit conversion to `int`.

An *interpolated_string_expression* is classified as a value. If it is immediately converted to `System.IFormattable` or `System.FormattableString` with an implicit interpolated string conversion ([Implicit interpolated string conversions](#)), the interpolated string expression has that type. Otherwise, it has the type `string`.

If the type of an interpolated string is `System.IFormattable` or `System.FormattableString`, the meaning is a call to `System.Runtime.CompilerServices.FormattableStringFactory.Create`. If the type is `string`, the meaning of the expression is a call to `string.Format`. In both cases, the argument list of the call consists of a format string literal with placeholders for each interpolation, and an argument for each expression corresponding to the placeholders.

The format string literal is constructed as follows, where `N` is the number of interpolations in the *interpolated_string_expression*.

- If an *interpolated_regular_string_whole* or an *interpolated_verbatim_string_whole* follows the `$` sign, then the format string literal is that token.
- Otherwise, the format string literal consists of:
 - First the *interpolated_regular_string_start* or *interpolated_verbatim_string_start*
 - Then for each number `I` from `0` to `N-1`:
 - The decimal representation of `I`
 - Then, if the corresponding *interpolation* has a *constant_expression*, a `,` (comma) followed by the decimal representation of the value of the *constant_expression*
 - Then the *interpolated_regular_string_mid*, *interpolated_regular_string_end*, *interpolated_verbatim_string_mid* or *interpolated_verbatim_string_end* immediately following the corresponding interpolation.

The subsequent arguments are simply the *expressions* from the *interpolations* (if any), in order.

TODO: examples.

Simple names

A *simple_name* consists of an identifier, optionally followed by a type argument list:

```
simple_name
: identifier type_argument_list?
;
```

A *simple_name* is either of the form `I` or of the form `I<A1, ..., Ak>`, where `I` is a single identifier and `<A1, ..., Ak>` is an optional *type_argument_list*. When no *type_argument_list* is specified, consider `K` to be zero. The *simple_name* is evaluated and classified as follows:

- If `K` is zero and the *simple_name* appears within a *block* and if the *block's* (or an enclosing *block's*) local variable declaration space ([Declarations](#)) contains a local variable, parameter or constant with name `I`, then the *simple_name* refers to that local variable, parameter or constant and is classified as a variable or value.
- If `K` is zero and the *simple_name* appears within the body of a generic method declaration and if that declaration includes a type parameter with name `I`, then the *simple_name* refers to that type parameter.
- Otherwise, for each instance type `T` ([The instance type](#)), starting with the instance type of the immediately enclosing type declaration and continuing with the instance type of each enclosing class or struct declaration (if any):
 - If `K` is zero and the declaration of `T` includes a type parameter with name `I`, then the *simple_name* refers to that type parameter.
 - Otherwise, if a member lookup ([Member lookup](#)) of `I` in `T` with `K` type arguments produces a match:
 - If `T` is the instance type of the immediately enclosing class or struct type and the lookup identifies one or more methods, the result is a method group with an associated instance expression of `this`. If a type argument list was specified, it is used in calling a generic method ([Method invocations](#)).
 - Otherwise, if `T` is the instance type of the immediately enclosing class or struct type, if the lookup identifies an instance member, and if the reference occurs within the body of an instance constructor, an instance method, or an instance accessor, the result is the same as a member access ([Member access](#)) of the form `this.I`. This can only happen when `K` is zero.
 - Otherwise, the result is the same as a member access ([Member access](#)) of the form `T.I` or `T.I<A1, ..., Ak>`. In this case, it is a binding-time error for the *simple_name* to refer to an instance member.
- Otherwise, for each namespace `N`, starting with the namespace in which the *simple_name* occurs, continuing with each enclosing namespace (if any), and ending with the global namespace, the following steps are evaluated until an entity is located:
 - If `K` is zero and `I` is the name of a namespace in `N`, then:
 - If the location where the *simple_name* occurs is enclosed by a namespace declaration for `N` and the namespace declaration contains an *extern_alias_directive* or *using_alias_directive* that associates the name `I` with a namespace or type, then the *simple_name* is ambiguous and a compile-time error occurs.
 - Otherwise, the *simple_name* refers to the namespace named `I` in `N`.
 - Otherwise, if `N` contains an accessible type having name `I` and `K` type parameters, then:
 - If `K` is zero and the location where the *simple_name* occurs is enclosed by a namespace declaration for `N` and the namespace declaration contains an *extern_alias_directive* or *using_alias_directive* that associates the name `I` with a namespace or type, then the

simple_name is ambiguous and a compile-time error occurs.

- Otherwise, the *namespace_or_type_name* refers to the type constructed with the given type arguments.
- Otherwise, if the location where the *simple_name* occurs is enclosed by a namespace declaration for **N**:
 - If **K** is zero and the namespace declaration contains an *extern_alias_directive* or *using_alias_directive* that associates the name **I** with an imported namespace or type, then the *simple_name* refers to that namespace or type.
 - Otherwise, if the namespaces and type declarations imported by the *using_namespace_directives* and *using_static_directives* of the namespace declaration contain exactly one accessible type or non-extension static member having name **I** and **K** type parameters, then the *simple_name* refers to that type or member constructed with the given type arguments.
 - Otherwise, if the namespaces and types imported by the *using_namespace_directives* of the namespace declaration contain more than one accessible type or non-extension-method static member having name **I** and **K** type parameters, then the *simple_name* is ambiguous and an error occurs.

Note that this entire step is exactly parallel to the corresponding step in the processing of a *namespace_or_type_name* ([Namespace and type names](#)).

- Otherwise, the *simple_name* is undefined and a compile-time error occurs.

Parenthesized expressions

A *parenthesized_expression* consists of an *expression* enclosed in parentheses.

```
parenthesized_expression
: '(' expression ')'
;
```

A *parenthesized_expression* is evaluated by evaluating the *expression* within the parentheses. If the *expression* within the parentheses denotes a namespace or type, a compile-time error occurs. Otherwise, the result of the *parenthesized_expression* is the result of the evaluation of the contained *expression*.

Member access

A *member_access* consists of a *primary_expression*, a *predefined_type*, or a *qualified_alias_member*, followed by a "." token, followed by an *identifier*, optionally followed by a *type_argument_list*.

```
member_access
: primary_expression '.' identifier type_argument_list?
| predefined_type '.' identifier type_argument_list?
| qualified_alias_member '.' identifier
;

predefined_type
: 'bool' | 'byte' | 'char' | 'decimal' | 'double' | 'float' | 'int' | 'long'
| 'object' | 'sbyte' | 'short' | 'string' | 'uint' | 'ulong' | 'ushort'
;
```

The *qualified_alias_member* production is defined in [Namespace alias qualifiers](#).

A *member_access* is either of the form **E.I** or of the form **E.I<A1, ..., Ak>**, where **E** is a primary-expression, **I** is a single identifier and **<A1, ..., Ak>** is an optional *type_argument_list*. When no *type_argument_list* is specified, consider **K** to be zero.

A *member_access* with a *primary_expression* of type `dynamic` is dynamically bound ([Dynamic binding](#)). In this case the compiler classifies the member access as a property access of type `dynamic`. The rules below to determine the meaning of the *member_access* are then applied at run-time, using the run-time type instead of the compile-time type of the *primary_expression*. If this run-time classification leads to a method group, then the member access must be the *primary_expression* of an *invocation_expression*.

The *member_access* is evaluated and classified as follows:

- If `K` is zero and `E` is a namespace and `E` contains a nested namespace with name `I`, then the result is that namespace.
- Otherwise, if `E` is a namespace and `E` contains an accessible type having name `I` and `K` type parameters, then the result is that type constructed with the given type arguments.
- If `E` is a *predefined_type* or a *primary_expression* classified as a type, if `E` is not a type parameter, and if a member lookup ([Member lookup](#)) of `I` in `E` with `K` type parameters produces a match, then `E.I` is evaluated and classified as follows:
 - If `I` identifies a type, then the result is that type constructed with the given type arguments.
 - If `I` identifies one or more methods, then the result is a method group with no associated instance expression. If a type argument list was specified, it is used in calling a generic method ([Method invocations](#)).
 - If `I` identifies a `static` property, then the result is a property access with no associated instance expression.
 - If `I` identifies a `static` field:
 - If the field is `readonly` and the reference occurs outside the static constructor of the class or struct in which the field is declared, then the result is a value, namely the value of the static field `I` in `E`.
 - Otherwise, the result is a variable, namely the static field `I` in `E`.
 - If `I` identifies a `static` event:
 - If the reference occurs within the class or struct in which the event is declared, and the event was declared without *event_accessor_declarations* ([Events](#)), then `E.I` is processed exactly as if `I` were a static field.
 - Otherwise, the result is an event access with no associated instance expression.
 - If `I` identifies a constant, then the result is a value, namely the value of that constant.
 - If `I` identifies an enumeration member, then the result is a value, namely the value of that enumeration member.
 - Otherwise, `E.I` is an invalid member reference, and a compile-time error occurs.
- If `E` is a property access, indexer access, variable, or value, the type of which is `T`, and a member lookup ([Member lookup](#)) of `I` in `T` with `K` type arguments produces a match, then `E.I` is evaluated and classified as follows:
 - First, if `E` is a property or indexer access, then the value of the property or indexer access is obtained ([Values of expressions](#)) and `E` is reclassified as a value.
 - If `I` identifies one or more methods, then the result is a method group with an associated instance expression of `E`. If a type argument list was specified, it is used in calling a generic method ([Method invocations](#)).
 - If `I` identifies an instance property,
 - If `E` is `this`, `I` identifies an automatically implemented property ([Automatically implemented properties](#)) without a setter, and the reference occurs within an instance constructor for a class or struct type `T`, then the result is a variable, namely the hidden backing field for the auto-property given by `I` in the instance of `T` given by `this`.
 - Otherwise, the result is a property access with an associated instance expression of `E`.

- If `T` is a *class_type* and `I` identifies an instance field of that *class_type*:
 - If the value of `E` is `null`, then a `System.NullReferenceException` is thrown.
 - Otherwise, if the field is `readonly` and the reference occurs outside an instance constructor of the class in which the field is declared, then the result is a value, namely the value of the field `I` in the object referenced by `E`.
 - Otherwise, the result is a variable, namely the field `I` in the object referenced by `E`.
- If `T` is a *struct_type* and `I` identifies an instance field of that *struct_type*:
 - If `E` is a value, or if the field is `readonly` and the reference occurs outside an instance constructor of the struct in which the field is declared, then the result is a value, namely the value of the field `I` in the struct instance given by `E`.
 - Otherwise, the result is a variable, namely the field `I` in the struct instance given by `E`.
- If `I` identifies an instance event:
 - If the reference occurs within the class or struct in which the event is declared, and the event was declared without *event_accessor_declarations* ([Events](#)), and the reference does not occur as the left-hand side of a `+=` or `-=` operator, then `E.I` is processed exactly as if `I` was an instance field.
 - Otherwise, the result is an event access with an associated instance expression of `E`.
- Otherwise, an attempt is made to process `E.I` as an extension method invocation ([Extension method invocations](#)). If this fails, `E.I` is an invalid member reference, and a binding-time error occurs.

Identical simple names and type names

In a member access of the form `E.I`, if `E` is a single identifier, and if the meaning of `E` as a *simple_name* ([Simple names](#)) is a constant, field, property, local variable, or parameter with the same type as the meaning of `E` as a *type_name* ([Namespace and type names](#)), then both possible meanings of `E` are permitted. The two possible meanings of `E.I` are never ambiguous, since `I` must necessarily be a member of the type `E` in both cases. In other words, the rule simply permits access to the static members and nested types of `E` where a compile-time error would otherwise have occurred. For example:

```
struct Color
{
    public static readonly Color White = new Color(...);
    public static readonly Color Black = new Color(...);

    public Color Complement() {...}
}

class A
{
    public Color Color;           // Field Color of type Color

    void F() {
        Color = Color.Black;      // References Color.Black static member
        Color = Color.Complement(); // Invokes Complement() on Color field
    }

    static void G() {
        Color c = Color.White;    // References Color.White static member
    }
}
```

Grammar ambiguities

The productions for *simple_name* ([Simple names](#)) and *member_access* ([Member access](#)) can give rise to ambiguities in the grammar for expressions. For example, the statement:

```
F(G<A,B>(7));
```

could be interpreted as a call to `F` with two arguments, `G < A` and `B > (7)`. Alternatively, it could be interpreted as a call to `F` with one argument, which is a call to a generic method `G` with two type arguments and one regular argument.

If a sequence of tokens can be parsed (in context) as a *simple_name* (Simple names), *member_access* (Member access), or *pointer_member_access* (Pointer member access) ending with a *type_argument_list* (Type arguments), the token immediately following the closing `>` token is examined. If it is one of

```
( ) ] } : ; , . ? == != | ^
```

then the *type_argument_list* is retained as part of the *simple_name*, *member_access* or *pointer_member_access* and any other possible parse of the sequence of tokens is discarded. Otherwise, the *type_argument_list* is not considered to be part of the *simple_name*, *member_access* or *pointer_member_access*, even if there is no other possible parse of the sequence of tokens. Note that these rules are not applied when parsing a *type_argument_list* in a *namespace_or_type_name* (Namespace and type names). The statement

```
F(G<A,B>(7));
```

will, according to this rule, be interpreted as a call to `F` with one argument, which is a call to a generic method `G` with two type arguments and one regular argument. The statements

```
F(G < A, B > 7);  
F(G < A, B >> 7);
```

will each be interpreted as a call to `F` with two arguments. The statement

```
x = F < A > +y;
```

will be interpreted as a less than operator, greater than operator, and unary plus operator, as if the statement had been written `x = (F < A) > (+y)`, instead of as a *simple_name* with a *type_argument_list* followed by a binary plus operator. In the statement

```
x = y is C<T> + z;
```

the tokens `C<T>` are interpreted as a *namespace_or_type_name* with a *type_argument_list*.

Invocation expressions

An *invocation_expression* is used to invoke a method.

```
invocation_expression  
    : primary_expression '(' argument_list? ')'  
    ;
```

An *invocation_expression* is dynamically bound (Dynamic binding) if at least one of the following holds:

- The *primary_expression* has compile-time type `dynamic`.
- At least one argument of the optional *argument_list* has compile-time type `dynamic` and the *primary_expression* does not have a delegate type.

In this case the compiler classifies the *invocation_expression* as a value of type `dynamic`. The rules below to determine the meaning of the *invocation_expression* are then applied at run-time, using the run-time type instead of the compile-time type of those of the *primary_expression* and arguments which have the compile-time type `dynamic`. If the *primary_expression* does not have compile-time type `dynamic`, then the method invocation undergoes a limited compile time check as described in [Compile-time checking of dynamic overload resolution](#).

The *primary_expression* of an *invocation_expression* must be a method group or a value of a *delegate_type*. If the *primary_expression* is a method group, the *invocation_expression* is a method invocation ([Method invocations](#)). If the *primary_expression* is a value of a *delegate_type*, the *invocation_expression* is a delegate invocation ([Delegate invocations](#)). If the *primary_expression* is neither a method group nor a value of a *delegate_type*, a binding-time error occurs.

The optional *argument_list* ([Argument lists](#)) provides values or variable references for the parameters of the method.

The result of evaluating an *invocation_expression* is classified as follows:

- If the *invocation_expression* invokes a method or delegate that returns `void`, the result is nothing. An expression that is classified as nothing is permitted only in the context of a *statement_expression* ([Expression statements](#)) or as the body of a *lambda_expression* ([Anonymous function expressions](#)). Otherwise a binding-time error occurs.
- Otherwise, the result is a value of the type returned by the method or delegate.

Method invocations

For a method invocation, the *primary_expression* of the *invocation_expression* must be a method group. The method group identifies the one method to invoke or the set of overloaded methods from which to choose a specific method to invoke. In the latter case, determination of the specific method to invoke is based on the context provided by the types of the arguments in the *argument_list*.

The binding-time processing of a method invocation of the form `M(A)`, where `M` is a method group (possibly including a *type_argument_list*), and `A` is an optional *argument_list*, consists of the following steps:

- The set of candidate methods for the method invocation is constructed. For each method `F` associated with the method group `M`:
 - If `F` is non-generic, `F` is a candidate when:
 - `M` has no type argument list, and
 - `F` is applicable with respect to `A` ([Applicable function member](#)).
 - If `F` is generic and `M` has no type argument list, `F` is a candidate when:
 - Type inference ([Type inference](#)) succeeds, inferring a list of type arguments for the call, and
 - Once the inferred type arguments are substituted for the corresponding method type parameters, all constructed types in the parameter list of `F` satisfy their constraints ([Satisfying constraints](#)), and the parameter list of `F` is applicable with respect to `A` ([Applicable function member](#)).
 - If `F` is generic and `M` includes a type argument list, `F` is a candidate when:
 - `F` has the same number of method type parameters as were supplied in the type argument list, and
 - Once the type arguments are substituted for the corresponding method type parameters, all constructed types in the parameter list of `F` satisfy their constraints ([Satisfying constraints](#)), and the parameter list of `F` is applicable with respect to `A` ([Applicable function member](#)).
- The set of candidate methods is reduced to contain only methods from the most derived types: For each method `C.F` in the set, where `C` is the type in which the method `F` is declared, all methods declared in a base type of `C` are removed from the set. Furthermore, if `C` is a class type other than `object`, all methods

declared in an interface type are removed from the set. (This latter rule only has affect when the method group was the result of a member lookup on a type parameter having an effective base class other than object and a non-empty effective interface set.)

- If the resulting set of candidate methods is empty, then further processing along the following steps are abandoned, and instead an attempt is made to process the invocation as an extension method invocation ([Extension method invocations](#)). If this fails, then no applicable methods exist, and a binding-time error occurs.
- The best method of the set of candidate methods is identified using the overload resolution rules of [Overload resolution](#). If a single best method cannot be identified, the method invocation is ambiguous, and a binding-time error occurs. When performing overload resolution, the parameters of a generic method are considered after substituting the type arguments (supplied or inferred) for the corresponding method type parameters.
- Final validation of the chosen best method is performed:
 - The method is validated in the context of the method group: If the best method is a static method, the method group must have resulted from a *simple_name* or a *member_access* through a type. If the best method is an instance method, the method group must have resulted from a *simple_name*, a *member_access* through a variable or value, or a *base_access*. If neither of these requirements is true, a binding-time error occurs.
 - If the best method is a generic method, the type arguments (supplied or inferred) are checked against the constraints ([Satisfying constraints](#)) declared on the generic method. If any type argument does not satisfy the corresponding constraint(s) on the type parameter, a binding-time error occurs.

Once a method has been selected and validated at binding-time by the above steps, the actual run-time invocation is processed according to the rules of function member invocation described in [Compile-time checking of dynamic overload resolution](#).

The intuitive effect of the resolution rules described above is as follows: To locate the particular method invoked by a method invocation, start with the type indicated by the method invocation and proceed up the inheritance chain until at least one applicable, accessible, non-override method declaration is found. Then perform type inference and overload resolution on the set of applicable, accessible, non-override methods declared in that type and invoke the method thus selected. If no method was found, try instead to process the invocation as an extension method invocation.

Extension method invocations

In a method invocation ([Invocations on boxed instances](#)) of one of the forms

```
expr . identifier ( )  
  
expr . identifier ( args )  
  
expr . identifier < typeargs > ( )  
  
expr . identifier < typeargs > ( args )
```

if the normal processing of the invocation finds no applicable methods, an attempt is made to process the construct as an extension method invocation. If *expr* or any of the *args* has compile-time type `dynamic`, extension methods will not apply.

The objective is to find the best *type_name* `c`, so that the corresponding static method invocation can take place:

```
C . identifier ( expr )  
  
C . identifier ( expr , args )  
  
C . identifier < typeargs > ( expr )  
  
C . identifier < typeargs > ( expr , args )
```

An extension method `Ci.Mj` is *eligible* if:

- `Ci` is a non-generic, non-nested class
- The name of `Mj` is *identifier*
- `Mj` is accessible and applicable when applied to the arguments as a static method as shown above
- An implicit identity, reference or boxing conversion exists from *expr* to the type of the first parameter of `Mj`.

The search for `C` proceeds as follows:

- Starting with the closest enclosing namespace declaration, continuing with each enclosing namespace declaration, and ending with the containing compilation unit, successive attempts are made to find a candidate set of extension methods:
 - If the given namespace or compilation unit directly contains non-generic type declarations `Ci` with eligible extension methods `Mj`, then the set of those extension methods is the candidate set.
 - If types `Ci` imported by *using_static_declarations* and directly declared in namespaces imported by *using_namespace_directives* in the given namespace or compilation unit directly contain eligible extension methods `Mj`, then the set of those extension methods is the candidate set.
- If no candidate set is found in any enclosing namespace declaration or compilation unit, a compile-time error occurs.
- Otherwise, overload resolution is applied to the candidate set as described in ([Overload resolution](#)). If no single best method is found, a compile-time error occurs.
- `C` is the type within which the best method is declared as an extension method.

Using `C` as a target, the method call is then processed as a static method invocation ([Compile-time checking of dynamic overload resolution](#)).

The preceding rules mean that instance methods take precedence over extension methods, that extension methods available in inner namespace declarations take precedence over extension methods available in outer namespace declarations, and that extension methods declared directly in a namespace take precedence over extension methods imported into that same namespace with a using namespace directive. For example:

```

public static class E
{
    public static void F(this object obj, int i) { }

    public static void F(this object obj, string s) { }
}

class A { }

class B
{
    public void F(int i) { }
}

class C
{
    public void F(object obj) { }
}

class X
{
    static void Test(A a, B b, C c) {
        a.F(1);           // E.F(object, int)
        a.F("hello");     // E.F(object, string)

        b.F(1);           // B.F(int)
        b.F("hello");     // E.F(object, string)

        c.F(1);           // C.F(object)
        c.F("hello");     // C.F(object)
    }
}

```

In the example, `B`'s method takes precedence over the first extension method, and `C`'s method takes precedence over both extension methods.

```

public static class C
{
    public static void F(this int i) { Console.WriteLine("C.F({0})", i); }
    public static void G(this int i) { Console.WriteLine("C.G({0})", i); }
    public static void H(this int i) { Console.WriteLine("C.H({0})", i); }
}

namespace N1
{
    public static class D
    {
        public static void F(this int i) { Console.WriteLine("D.F({0})", i); }
        public static void G(this int i) { Console.WriteLine("D.G({0})", i); }
    }
}

namespace N2
{
    using N1;

    public static class E
    {
        public static void F(this int i) { Console.WriteLine("E.F({0})", i); }
    }

    class Test
    {
        static void Main(string[] args)
        {
            1.F();
            2.G();
            3.H();
        }
    }
}

```

The output of this example is:

```

E.F(1)
D.G(2)
C.H(3)

```

`D.G` takes precedence over `C.G`, and `E.F` takes precedence over both `D.F` and `C.F`.

Delegate invocations

For a delegate invocation, the *primary_expression* of the *invocation_expression* must be a value of a *delegate_type*. Furthermore, considering the *delegate_type* to be a function member with the same parameter list as the *delegate_type*, the *delegate_type* must be applicable ([Applicable function member](#)) with respect to the *argument_list* of the *invocation_expression*.

The run-time processing of a delegate invocation of the form `D(A)`, where `D` is a *primary_expression* of a *delegate_type* and `A` is an optional *argument_list*, consists of the following steps:

- `D` is evaluated. If this evaluation causes an exception, no further steps are executed.
- The value of `D` is checked to be valid. If the value of `D` is `null`, a `System.NullReferenceException` is thrown and no further steps are executed.
- Otherwise, `D` is a reference to a delegate instance. Function member invocations ([Compile-time checking of dynamic overload resolution](#)) are performed on each of the callable entities in the invocation list of the delegate. For callable entities consisting of an instance and instance method, the instance for the invocation is the instance contained in the callable entity.

Element access

An *element_access* consists of a *primary_no_array_creation_expression*, followed by a "[" token, followed by an *argument_list*, followed by a "]" token. The *argument_list* consists of one or more *arguments*, separated by commas.

```
element_access
    : primary_no_array_creation_expression '[' expression_list ']'
    ;
```

The *argument_list* of an *element_access* is not allowed to contain `ref` or `out` arguments.

An *element_access* is dynamically bound ([Dynamic binding](#)) if at least one of the following holds:

- The *primary_no_array_creation_expression* has compile-time type `dynamic`.
- At least one expression of the *argument_list* has compile-time type `dynamic` and the *primary_no_array_creation_expression* does not have an array type.

In this case the compiler classifies the *element_access* as a value of type `dynamic`. The rules below to determine the meaning of the *element_access* are then applied at run-time, using the run-time type instead of the compile-time type of those of the *primary_no_array_creation_expression* and *argument_list* expressions which have the compile-time type `dynamic`. If the *primary_no_array_creation_expression* does not have compile-time type `dynamic`, then the element access undergoes a limited compile time check as described in [Compile-time checking of dynamic overload resolution](#).

If the *primary_no_array_creation_expression* of an *element_access* is a value of an *array_type*, the *element_access* is an array access ([Array access](#)). Otherwise, the *primary_no_array_creation_expression* must be a variable or value of a class, struct, or interface type that has one or more indexer members, in which case the *element_access* is an indexer access ([Indexer access](#)).

Array access

For an array access, the *primary_no_array_creation_expression* of the *element_access* must be a value of an *array_type*. Furthermore, the *argument_list* of an array access is not allowed to contain named arguments. The number of expressions in the *argument_list* must be the same as the rank of the *array_type*, and each expression must be of type `int`, `uint`, `long`, `ulong`, or must be implicitly convertible to one or more of these types.

The result of evaluating an array access is a variable of the element type of the array, namely the array element selected by the value(s) of the expression(s) in the *argument_list*.

The run-time processing of an array access of the form `P[A]`, where `P` is a *primary_no_array_creation_expression* of an *array_type* and `A` is an *argument_list*, consists of the following steps:

- `P` is evaluated. If this evaluation causes an exception, no further steps are executed.
- The index expressions of the *argument_list* are evaluated in order, from left to right. Following evaluation of each index expression, an implicit conversion ([Implicit conversions](#)) to one of the following types is performed: `int`, `uint`, `long`, `ulong`. The first type in this list for which an implicit conversion exists is chosen. For instance, if the index expression is of type `short` then an implicit conversion to `int` is performed, since implicit conversions from `short` to `int` and from `short` to `long` are possible. If evaluation of an index expression or the subsequent implicit conversion causes an exception, then no further index expressions are evaluated and no further steps are executed.
- The value of `P` is checked to be valid. If the value of `P` is `null`, a `System.NullReferenceException` is thrown and no further steps are executed.
- The value of each expression in the *argument_list* is checked against the actual bounds of each dimension of the array instance referenced by `P`. If one or more values are out of range, a

`System.IndexOutOfRangeException` is thrown and no further steps are executed.

- The location of the array element given by the index expression(s) is computed, and this location becomes the result of the array access.

Indexer access

For an indexer access, the *primary_no_array_creation_expression* of the *element_access* must be a variable or value of a class, struct, or interface type, and this type must implement one or more indexers that are applicable with respect to the *argument_list* of the *element_access*.

The binding-time processing of an indexer access of the form `P[A]`, where `P` is a *primary_no_array_creation_expression* of a class, struct, or interface type `T`, and `A` is an *argument_list*, consists of the following steps:

- The set of indexers provided by `T` is constructed. The set consists of all indexers declared in `T` or a base type of `T` that are not `override` declarations and are accessible in the current context ([Member access](#)).
- The set is reduced to those indexers that are applicable and not hidden by other indexers. The following rules are applied to each indexer `S.I` in the set, where `S` is the type in which the indexer `I` is declared:
 - If `I` is not applicable with respect to `A` ([Applicable function member](#)), then `I` is removed from the set.
 - If `I` is applicable with respect to `A` ([Applicable function member](#)), then all indexers declared in a base type of `S` are removed from the set.
 - If `I` is applicable with respect to `A` ([Applicable function member](#)) and `S` is a class type other than `object`, all indexers declared in an interface are removed from the set.
- If the resulting set of candidate indexers is empty, then no applicable indexers exist, and a binding-time error occurs.
- The best indexer of the set of candidate indexers is identified using the overload resolution rules of [Overload resolution](#). If a single best indexer cannot be identified, the indexer access is ambiguous, and a binding-time error occurs.
- The index expressions of the *argument_list* are evaluated in order, from left to right. The result of processing the indexer access is an expression classified as an indexer access. The indexer access expression references the indexer determined in the step above, and has an associated instance expression of `P` and an associated argument list of `A`.

Depending on the context in which it is used, an indexer access causes invocation of either the *get accessor* or the *set accessor* of the indexer. If the indexer access is the target of an assignment, the *set accessor* is invoked to assign a new value ([Simple assignment](#)). In all other cases, the *get accessor* is invoked to obtain the current value ([Values of expressions](#)).

This access

A *this_access* consists of the reserved word `this`.

```
this_access
: 'this'
;
```

A *this_access* is permitted only in the *block* of an instance constructor, an instance method, or an instance accessor. It has one of the following meanings:

- When `this` is used in a *primary_expression* within an instance constructor of a class, it is classified as a value. The type of the value is the instance type ([The instance type](#)) of the class within which the usage occurs, and the value is a reference to the object being constructed.
- When `this` is used in a *primary_expression* within an instance method or instance accessor of a class, it is classified as a value. The type of the value is the instance type ([The instance type](#)) of the class within which

the usage occurs, and the value is a reference to the object for which the method or accessor was invoked.

- When `this` is used in a *primary_expression* within an instance constructor of a struct, it is classified as a variable. The type of the variable is the instance type ([The instance type](#)) of the struct within which the usage occurs, and the variable represents the struct being constructed. The `this` variable of an instance constructor of a struct behaves exactly the same as an `out` parameter of the struct type—in particular, this means that the variable must be definitely assigned in every execution path of the instance constructor.
- When `this` is used in a *primary_expression* within an instance method or instance accessor of a struct, it is classified as a variable. The type of the variable is the instance type ([The instance type](#)) of the struct within which the usage occurs.
 - If the method or accessor is not an iterator ([Iterators](#)), the `this` variable represents the struct for which the method or accessor was invoked, and behaves exactly the same as a `ref` parameter of the struct type.
 - If the method or accessor is an iterator, the `this` variable represents a copy of the struct for which the method or accessor was invoked, and behaves exactly the same as a value parameter of the struct type.

Use of `this` in a *primary_expression* in a context other than the ones listed above is a compile-time error. In particular, it is not possible to refer to `this` in a static method, a static property accessor, or in a *variable_initializer* of a field declaration.

Base access

A *base_access* consists of the reserved word `base` followed by either a `.` token and an identifier or an *argument_list* enclosed in square brackets:

```
base_access
: 'base' '.' identifier
| 'base' '[' expression_list ']'
;
```

A *base_access* is used to access base class members that are hidden by similarly named members in the current class or struct. A *base_access* is permitted only in the *block* of an instance constructor, an instance method, or an instance accessor. When `base.I` occurs in a class or struct, `I` must denote a member of the base class of that class or struct. Likewise, when `base[E]` occurs in a class, an applicable indexer must exist in the base class.

At binding-time, *base_access* expressions of the form `base.I` and `base[E]` are evaluated exactly as if they were written `((B)this).I` and `((B)this)[E]`, where `B` is the base class of the class or struct in which the construct occurs. Thus, `base.I` and `base[E]` correspond to `this.I` and `this[E]`, except `this` is viewed as an instance of the base class.

When a *base_access* references a virtual function member (a method, property, or indexer), the determination of which function member to invoke at run-time ([Compile-time checking of dynamic overload resolution](#)) is changed. The function member that is invoked is determined by finding the most derived implementation ([Virtual methods](#)) of the function member with respect to `B` (instead of with respect to the run-time type of `this`, as would be usual in a non-base access). Thus, within an `override` of a `virtual` function member, a *base_access* can be used to invoke the inherited implementation of the function member. If the function member referenced by a *base_access* is abstract, a binding-time error occurs.

Postfix increment and decrement operators

```

post_increment_expression
    : primary_expression '++'
    ;

post_decrement_expression
    : primary_expression '--'
    ;

```

The operand of a postfix increment or decrement operation must be an expression classified as a variable, a property access, or an indexer access. The result of the operation is a value of the same type as the operand.

If the *primary_expression* has the compile-time type `dynamic` then the operator is dynamically bound ([Dynamic binding](#)), the *post_increment_expression* or *post_decrement_expression* has the compile-time type `dynamic` and the following rules are applied at run-time using the run-time type of the *primary_expression*.

If the operand of a postfix increment or decrement operation is a property or indexer access, the property or indexer must have both a `get` and a `set` accessor. If this is not the case, a binding-time error occurs.

Unary operator overload resolution ([Unary operator overload resolution](#)) is applied to select a specific operator implementation. Predefined `++` and `--` operators exist for the following types: `sbyte`, `byte`, `short`, `ushort`, `int`, `uint`, `long`, `ulong`, `char`, `float`, `double`, `decimal`, and any enum type. The predefined `++` operators return the value produced by adding 1 to the operand, and the predefined `--` operators return the value produced by subtracting 1 from the operand. In a `checked` context, if the result of this addition or subtraction is outside the range of the result type and the result type is an integral type or enum type, a `System.OverflowException` is thrown.

The run-time processing of a postfix increment or decrement operation of the form `x++` or `x--` consists of the following steps:

- If `x` is classified as a variable:
 - `x` is evaluated to produce the variable.
 - The value of `x` is saved.
 - The selected operator is invoked with the saved value of `x` as its argument.
 - The value returned by the operator is stored in the location given by the evaluation of `x`.
 - The saved value of `x` becomes the result of the operation.
- If `x` is classified as a property or indexer access:
 - The instance expression (if `x` is not `static`) and the argument list (if `x` is an indexer access) associated with `x` are evaluated, and the results are used in the subsequent `get` and `set` accessor invocations.
 - The `get` accessor of `x` is invoked and the returned value is saved.
 - The selected operator is invoked with the saved value of `x` as its argument.
 - The `set` accessor of `x` is invoked with the value returned by the operator as its `value` argument.
 - The saved value of `x` becomes the result of the operation.

The `++` and `--` operators also support prefix notation ([Prefix increment and decrement operators](#)). Typically, the result of `x++` or `x--` is the value of `x` before the operation, whereas the result of `++x` or `--x` is the value of `x` after the operation. In either case, `x` itself has the same value after the operation.

An `operator ++` or `operator --` implementation can be invoked using either postfix or prefix notation. It is not possible to have separate operator implementations for the two notations.

The new operator

The `new` operator is used to create new instances of types.

There are three forms of `new` expressions:

- Object creation expressions are used to create new instances of class types and value types.
- Array creation expressions are used to create new instances of array types.
- Delegate creation expressions are used to create new instances of delegate types.

The `new` operator implies creation of an instance of a type, but does not necessarily imply dynamic allocation of memory. In particular, instances of value types require no additional memory beyond the variables in which they reside, and no dynamic allocations occur when `new` is used to create instances of value types.

Object creation expressions

An *object_creation_expression* is used to create a new instance of a *class_type* or a *value_type*.

```
object_creation_expression
: 'new' type '(' argument_list? ')' object_or_collection_initializer?
| 'new' type object_or_collection_initializer
;

object_or_collection_initializer
: object_initializer
| collection_initializer
;
```

The *type* of an *object_creation_expression* must be a *class_type*, a *value_type* or a *type_parameter*. The *type* cannot be an `abstract` *class_type*.

The optional *argument_list* ([Argument lists](#)) is permitted only if the *type* is a *class_type* or a *struct_type*.

An object creation expression can omit the constructor argument list and enclosing parentheses provided it includes an object initializer or collection initializer. Omitting the constructor argument list and enclosing parentheses is equivalent to specifying an empty argument list.

Processing of an object creation expression that includes an object initializer or collection initializer consists of first processing the instance constructor and then processing the member or element initializations specified by the object initializer ([Object initializers](#)) or collection initializer ([Collection initializers](#)).

If any of the arguments in the optional *argument_list* has the compile-time type `dynamic` then the *object_creation_expression* is dynamically bound ([Dynamic binding](#)) and the following rules are applied at run-time using the run-time type of those arguments of the *argument_list* that have the compile time type `dynamic`. However, the object creation undergoes a limited compile time check as described in [Compile-time checking of dynamic overload resolution](#).

The binding-time processing of an *object_creation_expression* of the form `new T(A)`, where `T` is a *class_type* or a *value_type* and `A` is an optional *argument_list*, consists of the following steps:

- If `T` is a *value_type* and `A` is not present:
 - The *object_creation_expression* is a default constructor invocation. The result of the *object_creation_expression* is a value of type `T`, namely the default value for `T` as defined in [The System.ValueType type](#).
- Otherwise, if `T` is a *type_parameter* and `A` is not present:
 - If no value type constraint or constructor constraint ([Type parameter constraints](#)) has been specified for `T`, a binding-time error occurs.
 - The result of the *object_creation_expression* is a value of the run-time type that the type parameter has been bound to, namely the result of invoking the default constructor of that type. The run-time type may be a reference type or a value type.
- Otherwise, if `T` is a *class_type* or a *struct_type*:

- If `T` is an `abstract class_type`, a compile-time error occurs.
- The instance constructor to invoke is determined using the overload resolution rules of [Overload resolution](#). The set of candidate instance constructors consists of all accessible instance constructors declared in `T` which are applicable with respect to `A` ([Applicable function member](#)). If the set of candidate instance constructors is empty, or if a single best instance constructor cannot be identified, a binding-time error occurs.
- The result of the *object_creation_expression* is a value of type `T`, namely the value produced by invoking the instance constructor determined in the step above.
- Otherwise, the *object_creation_expression* is invalid, and a binding-time error occurs.

Even if the *object_creation_expression* is dynamically bound, the compile-time type is still `T`.

The run-time processing of an *object_creation_expression* of the form `new T(A)`, where `T` is *class_type* or a *struct_type* and `A` is an optional *argument_list*, consists of the following steps:

- If `T` is a *class_type*:
 - A new instance of class `T` is allocated. If there is not enough memory available to allocate the new instance, a `System.OutOfMemoryException` is thrown and no further steps are executed.
 - All fields of the new instance are initialized to their default values ([Default values](#)).
 - The instance constructor is invoked according to the rules of function member invocation ([Compile-time checking of dynamic overload resolution](#)). A reference to the newly allocated instance is automatically passed to the instance constructor and the instance can be accessed from within that constructor as `this`.
- If `T` is a *struct_type*:
 - An instance of type `T` is created by allocating a temporary local variable. Since an instance constructor of a *struct_type* is required to definitely assign a value to each field of the instance being created, no initialization of the temporary variable is necessary.
 - The instance constructor is invoked according to the rules of function member invocation ([Compile-time checking of dynamic overload resolution](#)). A reference to the newly allocated instance is automatically passed to the instance constructor and the instance can be accessed from within that constructor as `this`.

Object initializers

An *object initializer* specifies values for zero or more fields, properties or indexed elements of an object.

```

object_initializer
    : '{' member_initializer_list? '}'
    | '{' member_initializer_list ',' '}'
    ;

member_initializer_list
    : member_initializer (',' member_initializer)*
    ;

member_initializer
    : initializer_target '=' initializer_value
    ;

initializer_target
    : identifier
    | '[' argument_list ']'
    ;

initializer_value
    : expression
    | object_or_collection_initializer
    ;

```

An object initializer consists of a sequence of member initializers, enclosed by `{` and `}` tokens and separated by commas. Each *member_initializer* designates a target for the initialization. An *identifier* must name an accessible field or property of the object being initialized, whereas an *argument_list* enclosed in square brackets must specify arguments for an accessible indexer on the object being initialized. It is an error for an object initializer to include more than one member initializer for the same field or property.

Each *initializer_target* is followed by an equals sign and either an expression, an object initializer or a collection initializer. It is not possible for expressions within the object initializer to refer to the newly created object it is initializing.

A member initializer that specifies an expression after the equals sign is processed in the same way as an assignment ([Simple assignment](#)) to the target.

A member initializer that specifies an object initializer after the equals sign is a ***nested object initializer***, i.e. an initialization of an embedded object. Instead of assigning a new value to the field or property, the assignments in the nested object initializer are treated as assignments to members of the field or property. Nested object initializers cannot be applied to properties with a value type, or to read-only fields with a value type.

A member initializer that specifies a collection initializer after the equals sign is an initialization of an embedded collection. Instead of assigning a new collection to the target field, property or indexer, the elements given in the initializer are added to the collection referenced by the target. The target must be of a collection type that satisfies the requirements specified in [Collection initializers](#).

The arguments to an index initializer will always be evaluated exactly once. Thus, even if the arguments end up never getting used (e.g. because of an empty nested initializer), they will be evaluated for their side effects.

The following class represents a point with two coordinates:

```

public class Point
{
    int x, y;

    public int X { get { return x; } set { x = value; } }
    public int Y { get { return y; } set { y = value; } }
}

```

An instance of `Point` can be created and initialized as follows:

```
Point a = new Point { X = 0, Y = 1 };
```

which has the same effect as

```
Point __a = new Point();
__a.X = 0;
__a.Y = 1;
Point a = __a;
```

where `__a` is an otherwise invisible and inaccessible temporary variable. The following class represents a rectangle created from two points:

```
public class Rectangle
{
    Point p1, p2;

    public Point P1 { get { return p1; } set { p1 = value; } }
    public Point P2 { get { return p2; } set { p2 = value; } }
}
```

An instance of `Rectangle` can be created and initialized as follows:

```
Rectangle r = new Rectangle {
    P1 = new Point { X = 0, Y = 1 },
    P2 = new Point { X = 2, Y = 3 }
};
```

which has the same effect as

```
Rectangle __r = new Rectangle();
Point __p1 = new Point();
__p1.X = 0;
__p1.Y = 1;
__r.P1 = __p1;
Point __p2 = new Point();
__p2.X = 2;
__p2.Y = 3;
__r.P2 = __p2;
Rectangle r = __r;
```

where `__r`, `__p1` and `__p2` are temporary variables that are otherwise invisible and inaccessible.

If `Rectangle`'s constructor allocates the two embedded `Point` instances

```
public class Rectangle
{
    Point p1 = new Point();
    Point p2 = new Point();

    public Point P1 { get { return p1; } }
    public Point P2 { get { return p2; } }
}
```

the following construct can be used to initialize the embedded `Point` instances instead of assigning new instances:


```
Rectangle r = new Rectangle {
    P1 = { X = 0, Y = 1 },
    P2 = { X = 2, Y = 3 }
};
```

which has the same effect as

```
Rectangle __r = new Rectangle();
__r.P1.X = 0;
__r.P1.Y = 1;
__r.P2.X = 2;
__r.P2.Y = 3;
Rectangle r = __r;
```

Given an appropriate definition of C, the following example:

```
var c = new C {
    x = true,
    y = { a = "Hello" },
    z = { 1, 2, 3 },
    ["x"] = 5,
    [0,0] = { "a", "b" },
    [1,2] = {}
};
```

is equivalent to this series of assignments:

```
C __c = new C();
__c.x = true;
__c.y.a = "Hello";
__c.z.Add(1);
__c.z.Add(2);
__c.z.Add(3);
string __i1 = "x";
__c[__i1] = 5;
int __i2 = 0, __i3 = 0;
__c[__i2,__i3].Add("a");
__c[__i2,__i3].Add("b");
int __i4 = 1, __i5 = 2;
var c = __c;
```

where `__c`, etc., are generated variables that are invisible and inaccessible to the source code. Note that the arguments for `[0,0]` are evaluated only once, and the arguments for `[1,2]` are evaluated once even though they are never used.

Collection initializers

A collection initializer specifies the elements of a collection.

```

collection_initializer
    : '{' element_initializer_list '}'
    | '{' element_initializer_list ',' '}'
    ;

element_initializer_list
    : element_initializer (',' element_initializer)*
    ;

element_initializer
    : non_assignment_expression
    | '{' expression_list '}'
    ;

expression_list
    : expression (',' expression)*
    ;

```

A collection initializer consists of a sequence of element initializers, enclosed by `{` and `}` tokens and separated by commas. Each element initializer specifies an element to be added to the collection object being initialized, and consists of a list of expressions enclosed by `{` and `}` tokens and separated by commas. A single-expression element initializer can be written without braces, but cannot then be an assignment expression, to avoid ambiguity with member initializers. The *non_assignment_expression* production is defined in [Expression](#).

The following is an example of an object creation expression that includes a collection initializer:

```
List<int> digits = new List<int> { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
```

The collection object to which a collection initializer is applied must be of a type that implements `System.Collections.IEnumerable` or a compile-time error occurs. For each specified element in order, the collection initializer invokes an `Add` method on the target object with the expression list of the element initializer as argument list, applying normal member lookup and overload resolution for each invocation. Thus, the collection object must have an applicable instance or extension method with the name `Add` for each element initializer.

The following class represents a contact with a name and a list of phone numbers:

```

public class Contact
{
    string name;
    List<string> phoneNumbers = new List<string>();

    public string Name { get { return name; } set { name = value; } }

    public List<string> PhoneNumbers { get { return phoneNumbers; } }
}

```

A `List<Contact>` can be created and initialized as follows:

```
var contacts = new List<Contact> {
    new Contact {
        Name = "Chris Smith",
        PhoneNumbers = { "206-555-0101", "425-882-8080" }
    },
    new Contact {
        Name = "Bob Harris",
        PhoneNumbers = { "650-555-0199" }
    }
};
```

which has the same effect as

```
var __clist = new List<Contact>();
Contact __c1 = new Contact();
__c1.Name = "Chris Smith";
__c1.PhoneNumbers.Add("206-555-0101");
__c1.PhoneNumbers.Add("425-882-8080");
__clist.Add(__c1);
Contact __c2 = new Contact();
__c2.Name = "Bob Harris";
__c2.PhoneNumbers.Add("650-555-0199");
__clist.Add(__c2);
var contacts = __clist;
```

where `__clist`, `__c1` and `__c2` are temporary variables that are otherwise invisible and inaccessible.

Array creation expressions

An *array_creation_expression* is used to create a new instance of an *array_type*.

```
array_creation_expression
: 'new' non_array_type '[' expression_list '[' rank_specifier* array_initializer?
| 'new' array_type array_initializer
| 'new' rank_specifier array_initializer
;
```

An array creation expression of the first form allocates an array instance of the type that results from deleting each of the individual expressions from the expression list. For example, the array creation expression `new int[10,20]` produces an array instance of type `int[,]`, and the array creation expression `new int[10][,]` produces an array of type `int[][,]`. Each expression in the expression list must be of type `int`, `uint`, `long`, or `ulong`, or implicitly convertible to one or more of these types. The value of each expression determines the length of the corresponding dimension in the newly allocated array instance. Since the length of an array dimension must be nonnegative, it is a compile-time error to have a *constant_expression* with a negative value in the expression list.

Except in an unsafe context ([Unsafe contexts](#)), the layout of arrays is unspecified.

If an array creation expression of the first form includes an array initializer, each expression in the expression list must be a constant and the rank and dimension lengths specified by the expression list must match those of the array initializer.

In an array creation expression of the second or third form, the rank of the specified array type or rank specifier must match that of the array initializer. The individual dimension lengths are inferred from the number of elements in each of the corresponding nesting levels of the array initializer. Thus, the expression

```
new int[,] {{0, 1}, {2, 3}, {4, 5}}
```

exactly corresponds to

```
new int[3, 2] {{0, 1}, {2, 3}, {4, 5}}
```

An array creation expression of the third form is referred to as an *implicitly typed array creation expression*. It is similar to the second form, except that the element type of the array is not explicitly given, but determined as the best common type ([Finding the best common type of a set of expressions](#)) of the set of expressions in the array initializer. For a multidimensional array, i.e., one where the *rank_specifier* contains at least one comma, this set comprises all *expressions* found in nested *array_initializers*.

Array initializers are described further in [Array initializers](#).

The result of evaluating an array creation expression is classified as a value, namely a reference to the newly allocated array instance. The run-time processing of an array creation expression consists of the following steps:

- The dimension length expressions of the *expression_list* are evaluated in order, from left to right. Following evaluation of each expression, an implicit conversion ([Implicit conversions](#)) to one of the following types is performed: `int`, `uint`, `long`, `ulong`. The first type in this list for which an implicit conversion exists is chosen. If evaluation of an expression or the subsequent implicit conversion causes an exception, then no further expressions are evaluated and no further steps are executed.
- The computed values for the dimension lengths are validated as follows. If one or more of the values are less than zero, a `System.OverflowException` is thrown and no further steps are executed.
- An array instance with the given dimension lengths is allocated. If there is not enough memory available to allocate the new instance, a `System.OutOfMemoryException` is thrown and no further steps are executed.
- All elements of the new array instance are initialized to their default values ([Default values](#)).
- If the array creation expression contains an array initializer, then each expression in the array initializer is evaluated and assigned to its corresponding array element. The evaluations and assignments are performed in the order the expressions are written in the array initializer—in other words, elements are initialized in increasing index order, with the rightmost dimension increasing first. If evaluation of a given expression or the subsequent assignment to the corresponding array element causes an exception, then no further elements are initialized (and the remaining elements will thus have their default values).

An array creation expression permits instantiation of an array with elements of an array type, but the elements of such an array must be manually initialized. For example, the statement

```
int[][] a = new int[100][];
```

creates a single-dimensional array with 100 elements of type `int[]`. The initial value of each element is `null`. It is not possible for the same array creation expression to also instantiate the sub-arrays, and the statement

```
int[][] a = new int[100][5];           // Error
```

results in a compile-time error. Instantiation of the sub-arrays must instead be performed manually, as in

```
int[][] a = new int[100][];  
for (int i = 0; i < 100; i++) a[i] = new int[5];
```

When an array of arrays has a "rectangular" shape, that is when the sub-arrays are all of the same length, it is more efficient to use a multi-dimensional array. In the example above, instantiation of the array of arrays creates 101 objects—one outer array and 100 sub-arrays. In contrast,

```
int[,] = new int[100, 5];
```

creates only a single object, a two-dimensional array, and accomplishes the allocation in a single statement.

The following are examples of implicitly typed array creation expressions:

```
var a = new[] { 1, 10, 100, 1000 };           // int[]
var b = new[] { 1, 1.5, 2, 2.5 };             // double[]
var c = new[,] { { "hello", null }, { "world", "!" } }; // string[,]
var d = new[] { 1, "one", 2, "two" };          // Error
```

The last expression causes a compile-time error because neither `int` nor `string` is implicitly convertible to the other, and so there is no best common type. An explicitly typed array creation expression must be used in this case, for example specifying the type to be `object[]`. Alternatively, one of the elements can be cast to a common base type, which would then become the inferred element type.

Implicitly typed array creation expressions can be combined with anonymous object initializers ([Anonymous object creation expressions](#)) to create anonymously typed data structures. For example:

```
var contacts = new[] {
    new {
        Name = "Chris Smith",
        PhoneNumbers = new[] { "206-555-0101", "425-882-8080" }
    },
    new {
        Name = "Bob Harris",
        PhoneNumbers = new[] { "650-555-0199" }
    }
};
```

Delegate creation expressions

A *delegate_creation_expression* is used to create a new instance of a *delegate_type*.

```
delegate_creation_expression
: 'new' delegate_type '(' expression ')'
```

The argument of a delegate creation expression must be a method group, an anonymous function or a value of either the compile time type `dynamic` or a *delegate_type*. If the argument is a method group, it identifies the method and, for an instance method, the object for which to create a delegate. If the argument is an anonymous function it directly defines the parameters and method body of the delegate target. If the argument is a value it identifies a delegate instance of which to create a copy.

If the *expression* has the compile-time type `dynamic`, the *delegate_creation_expression* is dynamically bound ([Dynamic binding](#)), and the rules below are applied at run-time using the run-time type of the *expression*. Otherwise the rules are applied at compile-time.

The binding-time processing of a *delegate_creation_expression* of the form `new D(E)`, where `D` is a *delegate_type* and `E` is an *expression*, consists of the following steps:

- If `E` is a method group, the delegate creation expression is processed in the same way as a method group conversion ([Method group conversions](#)) from `E` to `D`.
- If `E` is an anonymous function, the delegate creation expression is processed in the same way as an

anonymous function conversion ([Anonymous function conversions](#)) from `E` to `D`.

- If `E` is a value, `E` must be compatible ([Delegate declarations](#)) with `D`, and the result is a reference to a newly created delegate of type `D` that refers to the same invocation list as `E`. If `E` is not compatible with `D`, a compile-time error occurs.

The run-time processing of a *delegate_creation_expression* of the form `new D(E)`, where `D` is a *delegate_type* and `E` is an *expression*, consists of the following steps:

- If `E` is a method group, the delegate creation expression is evaluated as a method group conversion ([Method group conversions](#)) from `E` to `D`.
- If `E` is an anonymous function, the delegate creation is evaluated as an anonymous function conversion from `E` to `D` ([Anonymous function conversions](#)).
- If `E` is a value of a *delegate_type*:
 - `E` is evaluated. If this evaluation causes an exception, no further steps are executed.
 - If the value of `E` is `null`, a `System.NullReferenceException` is thrown and no further steps are executed.
 - A new instance of the delegate type `D` is allocated. If there is not enough memory available to allocate the new instance, a `System.OutOfMemoryException` is thrown and no further steps are executed.
 - The new delegate instance is initialized with the same invocation list as the delegate instance given by `E`.

The invocation list of a delegate is determined when the delegate is instantiated and then remains constant for the entire lifetime of the delegate. In other words, it is not possible to change the target callable entities of a delegate once it has been created. When two delegates are combined or one is removed from another ([Delegate declarations](#)), a new delegate results; no existing delegate has its contents changed.

It is not possible to create a delegate that refers to a property, indexer, user-defined operator, instance constructor, destructor, or static constructor.

As described above, when a delegate is created from a method group, the formal parameter list and return type of the delegate determine which of the overloaded methods to select. In the example

```
delegate double DoubleFunc(double x);

class A
{
    DoubleFunc f = new DoubleFunc(Square);

    static float Square(float x) {
        return x * x;
    }

    static double Square(double x) {
        return x * x;
    }
}
```

the `A.f` field is initialized with a delegate that refers to the second `Square` method because that method exactly matches the formal parameter list and return type of `DoubleFunc`. Had the second `Square` method not been present, a compile-time error would have occurred.

Anonymous object creation expressions

An *anonymous_object_creation_expression* is used to create an object of an anonymous type.

```

anonymous_object_creation_expression
    : 'new' anonymous_object_initializer
    ;

anonymous_object_initializer
    : '{' member_declarator_list? '}'
    | '{' member_declarator_list ',' '}'
    ;

member_declarator_list
    : member_declarator (',' member_declarator)*
    ;

member_declarator
    : simple_name
    | member_access
    | base_access
    | null_conditional_member_access
    | identifier '=' expression
    ;

```

An anonymous object initializer declares an anonymous type and returns an instance of that type. An anonymous type is a nameless class type that inherits directly from `object`. The members of an anonymous type are a sequence of read-only properties inferred from the anonymous object initializer used to create an instance of the type. Specifically, an anonymous object initializer of the form

```
new { p1 = e1, p2 = e2, ..., pn = en }
```

declares an anonymous type of the form

```

class __Anonymous1
{
    private readonly T1 f1;
    private readonly T2 f2;
    ...
    private readonly Tn fn;

    public __Anonymous1(T1 a1, T2 a2, ..., Tn an) {
        f1 = a1;
        f2 = a2;
        ...
        fn = an;
    }

    public T1 p1 { get { return f1; } }
    public T2 p2 { get { return f2; } }
    ...
    public Tn pn { get { return fn; } }

    public override bool Equals(object __o) { ... }
    public override int GetHashCode() { ... }
}

```

where each `Tx` is the type of the corresponding expression `ex`. The expression used in a *member_declarator* must have a type. Thus, it is a compile-time error for an expression in a *member_declarator* to be null or an anonymous function. It is also a compile-time error for the expression to have an unsafe type.

The names of an anonymous type and of the parameter to its `Equals` method are automatically generated by the compiler and cannot be referenced in program text.

Within the same program, two anonymous object initializers that specify a sequence of properties of the same

names and compile-time types in the same order will produce instances of the same anonymous type.

In the example

```
var p1 = new { Name = "Lawnmower", Price = 495.00 };
var p2 = new { Name = "Shovel", Price = 26.95 };
p1 = p2;
```

the assignment on the last line is permitted because `p1` and `p2` are of the same anonymous type.

The `Equals` and `GetHashCode` methods on anonymous types override the methods inherited from `object`, and are defined in terms of the `Equals` and `GetHashCode` of the properties, so that two instances of the same anonymous type are equal if and only if all their properties are equal.

A member declarator can be abbreviated to a simple name ([Type inference](#)), a member access ([Compile-time checking of dynamic overload resolution](#)), a base access ([Base access](#)) or a null-conditional member access ([Null-conditional expressions as projection initializers](#)). This is called a *projection initializer* and is shorthand for a declaration of and assignment to a property with the same name. Specifically, member declarators of the forms

```
identifier
expr.identifier
```

are precisely equivalent to the following, respectively:

```
identifier = identifier
identifier = expr.identifier
```

Thus, in a projection initializer the *identifier* selects both the value and the field or property to which the value is assigned. Intuitively, a projection initializer projects not just a value, but also the name of the value.

The `typeof` operator

The `typeof` operator is used to obtain the `System.Type` object for a type.

```
typeof_expression
: 'typeof' '(' type ')'
| 'typeof' '(' unbound_type_name ')'
| 'typeof' '(' 'void' ')'
;

unbound_type_name
: identifier generic_dimension_specifier?
| identifier '::' identifier generic_dimension_specifier?
| unbound_type_name '.' identifier generic_dimension_specifier?
;

generic_dimension_specifier
: '<' comma* '>'
;

comma
: ','
;
```

The first form of *typeof_expression* consists of a `typeof` keyword followed by a parenthesized *type*. The result of an expression of this form is the `System.Type` object for the indicated type. There is only one `System.Type` object for any given type. This means that for a type `T`, `typeof(T) == typeof(T)` is always true. The *type* cannot

be `dynamic`.

The second form of *typeof_expression* consists of a `typeof` keyword followed by a parenthesized *unbound_type_name*. An *unbound_type_name* is very similar to a *type_name* ([Namespace and type names](#)) except that an *unbound_type_name* contains *generic_dimension_specifiers* where a *type_name* contains *type_argument_lists*. When the operand of a *typeof_expression* is a sequence of tokens that satisfies the grammars of both *unbound_type_name* and *type_name*, namely when it contains neither a *generic_dimension_specifier* nor a *type_argument_list*, the sequence of tokens is considered to be a *type_name*. The meaning of an *unbound_type_name* is determined as follows:

- Convert the sequence of tokens to a *type_name* by replacing each *generic_dimension_specifier* with a *type_argument_list* having the same number of commas and the keyword `object` as each *type_argument*.
- Evaluate the resulting *type_name*, while ignoring all type parameter constraints.
- The *unbound_type_name* resolves to the unbound generic type associated with the resulting constructed type ([Bound and unbound types](#)).

The result of the *typeof_expression* is the `System.Type` object for the resulting unbound generic type.

The third form of *typeof_expression* consists of a `typeof` keyword followed by a parenthesized `void` keyword. The result of an expression of this form is the `System.Type` object that represents the absence of a type. The type object returned by `typeof(void)` is distinct from the type object returned for any type. This special type object is useful in class libraries that allow reflection onto methods in the language, where those methods wish to have a way to represent the return type of any method, including void methods, with an instance of `System.Type`.

The `typeof` operator can be used on a type parameter. The result is the `System.Type` object for the run-time type that was bound to the type parameter. The `typeof` operator can also be used on a constructed type or an unbound generic type ([Bound and unbound types](#)). The `System.Type` object for an unbound generic type is not the same as the `System.Type` object of the instance type. The instance type is always a closed constructed type at run-time so its `System.Type` object depends on the run-time type arguments in use, while the unbound generic type has no type arguments.

The example

```

using System;

class X<T>
{
    public static void PrintTypes() {
        Type[] t = {
            typeof(int),
            typeof(System.Int32),
            typeof(string),
            typeof(double[]),
            typeof(void),
            typeof(T),
            typeof(X<T>),
            typeof(X<X<T>>),
            typeof(X<>)
        };
        for (int i = 0; i < t.Length; i++) {
            Console.WriteLine(t[i]);
        }
    }
}

class Test
{
    static void Main() {
        X<int>.PrintTypes();
    }
}

```

produces the following output:

```

System.Int32
System.Int32
System.String
System.Double[]
System.Void
System.Int32
X`1[System.Int32]
X`1[X`1[System.Int32]]
X`1[T]

```

Note that `int` and `System.Int32` are the same type.

Also note that the result of `typeof(X<>)` does not depend on the type argument but the result of `typeof(X<T>)` does.

The checked and unchecked operators

The `checked` and `unchecked` operators are used to control the *overflow checking context* for integral-type arithmetic operations and conversions.

```

checked_expression
: 'checked' '(' expression ')'
;

unchecked_expression
: 'unchecked' '(' expression ')'
;

```

The `checked` operator evaluates the contained expression in a checked context, and the `unchecked` operator evaluates the contained expression in an unchecked context. A *checked_expression* or *unchecked_expression* corresponds exactly to a *parenthesized_expression* ([Parenthesized expressions](#)), except that the contained

expression is evaluated in the given overflow checking context.

The overflow checking context can also be controlled through the `checked` and `unchecked` statements ([The checked and unchecked statements](#)).

The following operations are affected by the overflow checking context established by the `checked` and `unchecked` operators and statements:

- The predefined `++` and `--` unary operators ([Postfix increment and decrement operators](#) and [Prefix increment and decrement operators](#)), when the operand is of an integral type.
- The predefined `-` unary operator ([Unary minus operator](#)), when the operand is of an integral type.
- The predefined `+`, `-`, `*`, and `/` binary operators ([Arithmetic operators](#)), when both operands are of integral types.
- Explicit numeric conversions ([Explicit numeric conversions](#)) from one integral type to another integral type, or from `float` or `double` to an integral type.

When one of the above operations produce a result that is too large to represent in the destination type, the context in which the operation is performed controls the resulting behavior:

- In a `checked` context, if the operation is a constant expression ([Constant expressions](#)), a compile-time error occurs. Otherwise, when the operation is performed at run-time, a `System.OverflowException` is thrown.
- In an `unchecked` context, the result is truncated by discarding any high-order bits that do not fit in the destination type.

For non-constant expressions (expressions that are evaluated at run-time) that are not enclosed by any `checked` or `unchecked` operators or statements, the default overflow checking context is `unchecked` unless external factors (such as compiler switches and execution environment configuration) call for `checked` evaluation.

For constant expressions (expressions that can be fully evaluated at compile-time), the default overflow checking context is always `checked`. Unless a constant expression is explicitly placed in an `unchecked` context, overflows that occur during the compile-time evaluation of the expression always cause compile-time errors.

The body of an anonymous function is not affected by `checked` or `unchecked` contexts in which the anonymous function occurs.

In the example

```
class Test
{
    static readonly int x = 1000000;
    static readonly int y = 1000000;

    static int F() {
        return checked(x * y);    // Throws OverflowException
    }

    static int G() {
        return unchecked(x * y); // Returns -727379968
    }

    static int H() {
        return x * y;            // Depends on default
    }
}
```

no compile-time errors are reported since neither of the expressions can be evaluated at compile-time. At run-time, the `F` method throws a `System.OverflowException`, and the `G` method returns -727379968 (the lower 32 bits of the out-of-range result). The behavior of the `H` method depends on the default overflow checking

context for the compilation, but it is either the same as `F` or the same as `G`.

In the example

```
class Test
{
    const int x = 1000000;
    const int y = 1000000;

    static int F() {
        return checked(x * y);    // Compile error, overflow
    }

    static int G() {
        return unchecked(x * y); // Returns -727379968
    }

    static int H() {
        return x * y;            // Compile error, overflow
    }
}
```

the overflows that occur when evaluating the constant expressions in `F` and `H` cause compile-time errors to be reported because the expressions are evaluated in a `checked` context. An overflow also occurs when evaluating the constant expression in `G`, but since the evaluation takes place in an `unchecked` context, the overflow is not reported.

The `checked` and `unchecked` operators only affect the overflow checking context for those operations that are textually contained within the "`(`" and "`)`" tokens. The operators have no effect on function members that are invoked as a result of evaluating the contained expression. In the example

```
class Test
{
    static int Multiply(int x, int y) {
        return x * y;
    }

    static int F() {
        return checked(Multiply(1000000, 1000000));
    }
}
```

the use of `checked` in `F` does not affect the evaluation of `x * y` in `Multiply`, so `x * y` is evaluated in the default overflow checking context.

The `unchecked` operator is convenient when writing constants of the signed integral types in hexadecimal notation. For example:

```
class Test
{
    public const int AllBits = unchecked((int)0xFFFFFFFF);

    public const int HighBit = unchecked((int)0x80000000);
}
```

Both of the hexadecimal constants above are of type `uint`. Because the constants are outside the `int` range, without the `unchecked` operator, the casts to `int` would produce compile-time errors.

The `checked` and `unchecked` operators and statements allow programmers to control certain aspects of some

numeric calculations. However, the behavior of some numeric operators depends on their operands' data types. For example, multiplying two decimals always results in an exception on overflow even within an explicitly `unchecked` construct. Similarly, multiplying two floats never results in an exception on overflow even within an explicitly `checked` construct. In addition, other operators are never affected by the mode of checking, whether default or explicit.

Default value expressions

A default value expression is used to obtain the default value ([Default values](#)) of a type. Typically a default value expression is used for type parameters, since it may not be known if the type parameter is a value type or a reference type. (No conversion exists from the `null` literal to a type parameter unless the type parameter is known to be a reference type.)

```
default_value_expression
    : 'default' '(' type ')'
    ;
```

If the *type* in a *default_value_expression* evaluates at run-time to a reference type, the result is `null` converted to that type. If the *type* in a *default_value_expression* evaluates at run-time to a value type, the result is the *value_type*'s default value ([Default constructors](#)).

A *default_value_expression* is a constant expression ([Constant expressions](#)) if the type is a reference type or a type parameter that is known to be a reference type ([Type parameter constraints](#)). In addition, a *default_value_expression* is a constant expression if the type is one of the following value types: `sbyte`, `byte`, `short`, `ushort`, `int`, `uint`, `long`, `ulong`, `char`, `float`, `double`, `decimal`, `bool`, or any enumeration type.

Nameof expressions

A *nameof_expression* is used to obtain the name of a program entity as a constant string.

```
nameof_expression
    : 'nameof' '(' named_entity ')'
    ;

named_entity
    : simple_name
    | named_entity_target '.' identifier type_argument_list?
    ;

named_entity_target
    : 'this'
    | 'base'
    | named_entity
    | predefined_type
    | qualified_alias_member
    ;
```

Grammatically speaking, the *named_entity* operand is always an expression. Because `nameof` is not a reserved keyword, a *nameof* expression is always syntactically ambiguous with an invocation of the simple name `nameof`. For compatibility reasons, if a name lookup ([Simple names](#)) of the name `nameof` succeeds, the expression is treated as an *invocation_expression* -- regardless of whether the invocation is legal. Otherwise it is a *nameof_expression*.

The meaning of the *named_entity* of a *nameof_expression* is the meaning of it as an expression; that is, either as a *simple_name*, a *base_access* or a *member_access*. However, where the lookup described in [Simple names](#) and [Member access](#) results in an error because an instance member was found in a static context, a *nameof_expression* produces no such error.

It is a compile-time error for a *named_entity* designating a method group to have a *type_argument_list*. It is a

compile time error for a *named_entity_target* to have the type `dynamic`.

A *nameof_expression* is a constant expression of type `string`, and has no effect at runtime. Specifically, its *named_entity* is not evaluated, and is ignored for the purposes of definite assignment analysis ([General rules for simple expressions](#)). Its value is the last identifier of the *named_entity* before the optional final *type_argument_list*, transformed in the following way:

- The prefix "`@`", if used, is removed.
- Each *unicode_escape_sequence* is transformed into its corresponding Unicode character.
- Any *formatting_characters* are removed.

These are the same transformations applied in [Identifiers](#) when testing equality between identifiers.

TODO: examples

Anonymous method expressions

An *anonymous_method_expression* is one of two ways of defining an anonymous function. These are further described in [Anonymous function expressions](#).

Unary operators

The `?`, `+`, `-`, `!`, `~`, `++`, `--`, `cast`, and `await` operators are called the unary operators.

```
unary_expression
: primary_expression
| null_conditional_expression
| '+' unary_expression
| '-' unary_expression
| '!' unary_expression
| '~' unary_expression
| pre_increment_expression
| pre_decrement_expression
| cast_expression
| await_expression
| unary_expression_unsafe
;
```

If the operand of a *unary_expression* has the compile-time type `dynamic`, it is dynamically bound ([Dynamic binding](#)). In this case the compile-time type of the *unary_expression* is `dynamic`, and the resolution described below will take place at run-time using the run-time type of the operand.

Null-conditional operator

The null-conditional operator applies a list of operations to its operand only if that operand is non-null. Otherwise the result of applying the operator is `null`.

```
null_conditional_expression
: primary_expression null_conditional_operations
;

null_conditional_operations
: null_conditional_operations? '?' '.' identifier type_argument_list?
| null_conditional_operations? '?' '[' argument_list ']'
| null_conditional_operations '.' identifier type_argument_list?
| null_conditional_operations '[' argument_list ']'
| null_conditional_operations '(' argument_list? ')'
;
```

The list of operations can include member access and element access operations (which may themselves be

null-conditional), as well as invocation.

For example, the expression `a.b?[0]?.c()` is a *null_conditional_expression* with a *primary_expression* `a.b` and *null_conditional_operations* `?[0]` (null-conditional element access), `?.c` (null-conditional member access) and `()` (invocation).

For a *null_conditional_expression* `E` with a *primary_expression* `P`, let `E0` be the expression obtained by textually removing the leading `?` from each of the *null_conditional_operations* of `E` that have one. Conceptually, `E0` is the expression that will be evaluated if none of the null checks represented by the `?`s do find a `null`.

Also, let `E1` be the expression obtained by textually removing the leading `?` from just the first of the *null_conditional_operations* in `E`. This may lead to a *primary-expression* (if there was just one `?`) or to another *null_conditional_expression*.

For example, if `E` is the expression `a.b?[0]?.c()`, then `E0` is the expression `a.b[0].c()` and `E1` is the expression `a.b[0]?.c()`.

If `E0` is classified as nothing, then `E` is classified as nothing. Otherwise `E` is classified as a value.

`E0` and `E1` are used to determine the meaning of `E`:

- If `E` occurs as a *statement_expression* the meaning of `E` is the same as the statement

```
if ((object)P != null) E1;
```

except that `P` is evaluated only once.

- Otherwise, if `E0` is classified as nothing a compile-time error occurs.
- Otherwise, let `T0` be the type of `E0`.
 - If `T0` is a type parameter that is not known to be a reference type or a non-nullable value type, a compile-time error occurs.
 - If `T0` is a non-nullable value type, then the type of `E` is `T0?`, and the meaning of `E` is the same as

```
((object)P == null) ? (T0?)null : E1
```

except that `P` is evaluated only once.

- Otherwise the type of `E` is `T0`, and the meaning of `E` is the same as

```
((object)P == null) ? null : E1
```

except that `P` is evaluated only once.

If `E1` is itself a *null_conditional_expression*, then these rules are applied again, nesting the tests for `null` until there are no further `?`'s, and the expression has been reduced all the way down to the *primary-expression* `E0`.

For example, if the expression `a.b?[0]?.c()` occurs as a *statement-expression*, as in the statement:

```
a.b?[0]?.c();
```

its meaning is equivalent to:

```
if (a.b != null) a.b[0]?.c();
```

which again is equivalent to:

```
if (a.b != null) if (a.b[0] != null) a.b[0].c();
```

Except that `a.b` and `a.b[0]` are evaluated only once.

If it occurs in a context where its value is used, as in:

```
var x = a.b?[0]?.c();
```

and assuming that the type of the final invocation is not a non-nullable value type, its meaning is equivalent to:

```
var x = (a.b == null) ? null : (a.b[0] == null) ? null : a.b[0].c();
```

except that `a.b` and `a.b[0]` are evaluated only once.

Null-conditional expressions as projection initializers

A null-conditional expression is only allowed as a *member_declarator* in an *anonymous_object_creation_expression* ([Anonymous object creation expressions](#)) if it ends with an (optionally null-conditional) member access. Grammatically, this requirement can be expressed as:

```
null_conditional_member_access
: primary_expression null_conditional_operations? '?' '.' identifier type_argument_list?
| primary_expression null_conditional_operations '.' identifier type_argument_list?
;
```

This is a special case of the grammar for *null_conditional_expression* above. The production for *member_declarator* in [Anonymous object creation expressions](#) then includes only *null_conditional_member_access*.

Null-conditional expressions as statement expressions

A null-conditional expression is only allowed as a *statement_expression* ([Expression statements](#)) if it ends with an invocation. Grammatically, this requirement can be expressed as:

```
null_conditional_invocation_expression
: primary_expression null_conditional_operations '(' argument_list? ')'
;
```

This is a special case of the grammar for *null_conditional_expression* above. The production for *statement_expression* in [Expression statements](#) then includes only *null_conditional_invocation_expression*.

Unary plus operator

For an operation of the form `+x`, unary operator overload resolution ([Unary operator overload resolution](#)) is applied to select a specific operator implementation. The operand is converted to the parameter type of the selected operator, and the type of the result is the return type of the operator. The predefined unary plus operators are:


```
int operator +(int x);
uint operator +(uint x);
long operator +(long x);
ulong operator +(ulong x);
float operator +(float x);
double operator +(double x);
decimal operator +(decimal x);
```

For each of these operators, the result is simply the value of the operand.

Unary minus operator

For an operation of the form `-x`, unary operator overload resolution ([Unary operator overload resolution](#)) is applied to select a specific operator implementation. The operand is converted to the parameter type of the selected operator, and the type of the result is the return type of the operator. The predefined negation operators are:

- Integer negation:

```
int operator -(int x);
long operator -(long x);
```

The result is computed by subtracting `x` from zero. If the value of `x` is the smallest representable value of the operand type (-2^{31} for `int` or -2^{63} for `long`), then the mathematical negation of `x` is not representable within the operand type. If this occurs within a `checked` context, a `System.OverflowException` is thrown; if it occurs within an `unchecked` context, the result is the value of the operand and the overflow is not reported.

If the operand of the negation operator is of type `uint`, it is converted to type `long`, and the type of the result is `long`. An exception is the rule that permits the `int` value -2^{31} to be written as a decimal integer literal ([Integer literals](#)).

If the operand of the negation operator is of type `ulong`, a compile-time error occurs. An exception is the rule that permits the `long` value -2^{63} to be written as a decimal integer literal ([Integer literals](#)).

- Floating-point negation:

```
float operator -(float x);
double operator -(double x);
```

The result is the value of `x` with its sign inverted. If `x` is NaN, the result is also NaN.

- Decimal negation:

```
decimal operator -(decimal x);
```

The result is computed by subtracting `x` from zero. Decimal negation is equivalent to using the unary minus operator of type `System.Decimal`.

Logical negation operator

For an operation of the form `!x`, unary operator overload resolution ([Unary operator overload resolution](#)) is applied to select a specific operator implementation. The operand is converted to the parameter type of the selected operator, and the type of the result is the return type of the operator. Only one predefined logical negation operator exists:

```
bool operator !(bool x);
```

This operator computes the logical negation of the operand: If the operand is `true`, the result is `false`. If the operand is `false`, the result is `true`.

Bitwise complement operator

For an operation of the form `~x`, unary operator overload resolution ([Unary operator overload resolution](#)) is applied to select a specific operator implementation. The operand is converted to the parameter type of the selected operator, and the type of the result is the return type of the operator. The predefined bitwise complement operators are:

```
int operator ~(int x);
uint operator ~(uint x);
long operator ~(long x);
ulong operator ~(ulong x);
```

For each of these operators, the result of the operation is the bitwise complement of `x`.

Every enumeration type `E` implicitly provides the following bitwise complement operator:

```
E operator ~(E x);
```

The result of evaluating `~x`, where `x` is an expression of an enumeration type `E` with an underlying type `U`, is exactly the same as evaluating `(E)(~(U)x)`, except that the conversion to `E` is always performed as if in an `unchecked` context ([The checked and unchecked operators](#)).

Prefix increment and decrement operators

```
pre_increment_expression
: '++' unary_expression
;

pre_decrement_expression
: '--' unary_expression
;
```

The operand of a prefix increment or decrement operation must be an expression classified as a variable, a property access, or an indexer access. The result of the operation is a value of the same type as the operand.

If the operand of a prefix increment or decrement operation is a property or indexer access, the property or indexer must have both a `get` and a `set` accessor. If this is not the case, a binding-time error occurs.

Unary operator overload resolution ([Unary operator overload resolution](#)) is applied to select a specific operator implementation. Predefined `++` and `--` operators exist for the following types: `sbyte`, `byte`, `short`, `ushort`, `int`, `uint`, `long`, `ulong`, `char`, `float`, `double`, `decimal`, and any enum type. The predefined `++` operators return the value produced by adding 1 to the operand, and the predefined `--` operators return the value produced by subtracting 1 from the operand. In a `checked` context, if the result of this addition or subtraction is outside the range of the result type and the result type is an integral type or enum type, a `System.OverflowException` is thrown.

The run-time processing of a prefix increment or decrement operation of the form `++x` or `--x` consists of the following steps:

- If `x` is classified as a variable:

- `x` is evaluated to produce the variable.
- The selected operator is invoked with the value of `x` as its argument.
- The value returned by the operator is stored in the location given by the evaluation of `x`.
- The value returned by the operator becomes the result of the operation.
- If `x` is classified as a property or indexer access:
 - The instance expression (if `x` is not `static`) and the argument list (if `x` is an indexer access) associated with `x` are evaluated, and the results are used in the subsequent `get` and `set` accessor invocations.
 - The `get` accessor of `x` is invoked.
 - The selected operator is invoked with the value returned by the `get` accessor as its argument.
 - The `set` accessor of `x` is invoked with the value returned by the operator as its `value` argument.
 - The value returned by the operator becomes the result of the operation.

The `++` and `--` operators also support postfix notation ([Postfix increment and decrement operators](#)). Typically, the result of `x++` or `x--` is the value of `x` before the operation, whereas the result of `++x` or `--x` is the value of `x` after the operation. In either case, `x` itself has the same value after the operation.

An `operator++` or `operator--` implementation can be invoked using either postfix or prefix notation. It is not possible to have separate operator implementations for the two notations.

Cast expressions

A *cast_expression* is used to explicitly convert an expression to a given type.

```
cast_expression
: '(' type ')' unary_expression
;
```

A *cast_expression* of the form `(T)E`, where `T` is a *type* and `E` is a *unary_expression*, performs an explicit conversion ([Explicit conversions](#)) of the value of `E` to type `T`. If no explicit conversion exists from `E` to `T`, a binding-time error occurs. Otherwise, the result is the value produced by the explicit conversion. The result is always classified as a value, even if `E` denotes a variable.

The grammar for a *cast_expression* leads to certain syntactic ambiguities. For example, the expression `(x)-y` could either be interpreted as a *cast_expression* (a cast of `-y` to type `x`) or as an *additive_expression* combined with a *parenthesized_expression* (which computes the value `x - y`).

To resolve *cast_expression* ambiguities, the following rule exists: A sequence of one or more *tokens* ([White space](#)) enclosed in parentheses is considered the start of a *cast_expression* only if at least one of the following are true:

- The sequence of tokens is correct grammar for a *type*, but not for an *expression*.
- The sequence of tokens is correct grammar for a *type*, and the token immediately following the closing parentheses is the token `~`, the token `!`, the token `(`, an *identifier* ([Unicode character escape sequences](#)), a *literal* ([Literals](#)), or any *keyword* ([Keywords](#)) except `as` and `is`.

The term "correct grammar" above means only that the sequence of tokens must conform to the particular grammatical production. It specifically does not consider the actual meaning of any constituent identifiers. For example, if `x` and `y` are identifiers, then `x.y` is correct grammar for a type, even if `x.y` doesn't actually denote a type.

From the disambiguation rule it follows that, if `x` and `y` are identifiers, `(x)y`, `(x)(y)`, and `(x)(-y)` are *cast_expressions*, but `(x)-y` is not, even if `x` identifies a type. However, if `x` is a keyword that identifies a predefined type (such as `int`), then all four forms are *cast_expressions* (because such a keyword could not

possibly be an expression by itself).

Await expressions

The `await` operator is used to suspend evaluation of the enclosing `async` function until the asynchronous operation represented by the operand has completed.

```
await_expression
: 'await' unary_expression
;
```

An *await_expression* is only allowed in the body of an `async` function ([Async functions](#)). Within the nearest enclosing `async` function, an *await_expression* may not occur in these places:

- Inside a nested (non-`async`) anonymous function
- Inside the block of a *lock_statement*
- In an unsafe context

Note that an *await_expression* cannot occur in most places within a *query_expression*, because those are syntactically transformed to use non-`async` lambda expressions.

Inside of an `async` function, `await` cannot be used as an identifier. There is therefore no syntactic ambiguity between *await-expressions* and various expressions involving identifiers. Outside of `async` functions, `await` acts as a normal identifier.

The operand of an *await_expression* is called the *task*. It represents an asynchronous operation that may or may not be complete at the time the *await_expression* is evaluated. The purpose of the `await` operator is to suspend execution of the enclosing `async` function until the awaited task is complete, and then obtain its outcome.

Awaitable expressions

The task of an *await* expression is required to be *awaitable*. An expression `t` is awaitable if one of the following holds:

- `t` is of compile time type `dynamic`
- `t` has an accessible instance or extension method called `GetAwaiter` with no parameters and no type parameters, and a return type `A` for which all of the following hold:
 - `A` implements the interface `System.Runtime.CompilerServices.INotifyCompletion` (hereafter known as `INotifyCompletion` for brevity)
 - `A` has an accessible, readable instance property `IsCompleted` of type `bool`
 - `A` has an accessible instance method `GetResult` with no parameters and no type parameters

The purpose of the `GetAwaiter` method is to obtain an *awaiter* for the task. The type `A` is called the *awaiter type* for the *await* expression.

The purpose of the `IsCompleted` property is to determine if the task is already complete. If so, there is no need to suspend evaluation.

The purpose of the `INotifyCompletion.OnCompleted` method is to sign up a "continuation" to the task; i.e. a delegate (of type `System.Action`) that will be invoked once the task is complete.

The purpose of the `GetResult` method is to obtain the outcome of the task once it is complete. This outcome may be successful completion, possibly with a result value, or it may be an exception which is thrown by the `GetResult` method.

Classification of await expressions

The expression `await t` is classified the same way as the expression `(t).GetAwaiter().GetResult()`. Thus, if the return type of `GetResult` is `void`, the *await_expression* is classified as nothing. If it has a non-void return type

`T`, the *await_expression* is classified as a value of type `T`.

Runtime evaluation of await expressions

At runtime, the expression `await t` is evaluated as follows:

- An awaiter `a` is obtained by evaluating the expression `(t).GetAwaiter()`.
- A `bool b` is obtained by evaluating the expression `(a).IsCompleted`.
- If `b` is `false` then evaluation depends on whether `a` implements the interface `System.Runtime.CompilerServices.ICriticalNotifyCompletion` (hereafter known as `ICriticalNotifyCompletion` for brevity). This check is done at binding time; i.e. at runtime if `a` has the compile time type `dynamic`, and at compile time otherwise. Let `r` denote the resumption delegate ([Async functions](#)):
 - If `a` does not implement `ICriticalNotifyCompletion`, then the expression `(a as INotifyCompletion).OnCompleted(r)` is evaluated.
 - If `a` does implement `ICriticalNotifyCompletion`, then the expression `(a as ICriticalNotifyCompletion).UnsafeOnCompleted(r)` is evaluated.
 - Evaluation is then suspended, and control is returned to the current caller of the async function.
- Either immediately after (if `b` was `true`), or upon later invocation of the resumption delegate (if `b` was `false`), the expression `(a).GetResult()` is evaluated. If it returns a value, that value is the result of the *await_expression*. Otherwise the result is nothing.

An awaiter's implementation of the interface methods `INotifyCompletion.OnCompleted` and `ICriticalNotifyCompletion.UnsafeOnCompleted` should cause the delegate `r` to be invoked at most once. Otherwise, the behavior of the enclosing async function is undefined.

Arithmetic operators

The `*`, `/`, `%`, `+`, and `-` operators are called the arithmetic operators.

```
multiplicative_expression
: unary_expression
| multiplicative_expression '*' unary_expression
| multiplicative_expression '/' unary_expression
| multiplicative_expression '%' unary_expression
;

additive_expression
: multiplicative_expression
| additive_expression '+' multiplicative_expression
| additive_expression '-' multiplicative_expression
;
```

If an operand of an arithmetic operator has the compile-time type `dynamic`, then the expression is dynamically bound ([Dynamic binding](#)). In this case the compile-time type of the expression is `dynamic`, and the resolution described below will take place at run-time using the run-time type of those operands that have the compile-time type `dynamic`.

Multiplication operator

For an operation of the form `x * y`, binary operator overload resolution ([Binary operator overload resolution](#)) is applied to select a specific operator implementation. The operands are converted to the parameter types of the selected operator, and the type of the result is the return type of the operator.

The predefined multiplication operators are listed below. The operators all compute the product of `x` and `y`.

- Integer multiplication:

```
int operator *(int x, int y);
uint operator *(uint x, uint y);
long operator *(long x, long y);
ulong operator *(ulong x, ulong y);
```

In a `checked` context, if the product is outside the range of the result type, a `System.OverflowException` is thrown. In an `unchecked` context, overflows are not reported and any significant high-order bits outside the range of the result type are discarded.

- Floating-point multiplication:

```
float operator *(float x, float y);
double operator *(double x, double y);
```

The product is computed according to the rules of IEEE 754 arithmetic. The following table lists the results of all possible combinations of nonzero finite values, zeros, infinities, and NaN's. In the table, `x` and `y` are positive finite values. `z` is the result of `x * y`. If the result is too large for the destination type, `z` is infinity. If the result is too small for the destination type, `z` is zero.

	+Y	-Y	+0	-0	+INF	-INF	NAN
+x	+z	-z	+0	-0	+inf	-inf	NaN
-x	-z	+z	-0	+0	-inf	+inf	NaN
+0	+0	-0	+0	-0	NaN	NaN	NaN
-0	-0	+0	-0	+0	NaN	NaN	NaN
+inf	+inf	-inf	NaN	NaN	+inf	-inf	NaN
-inf	-inf	+inf	NaN	NaN	-inf	+inf	NaN
NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN

- Decimal multiplication:

```
decimal operator *(decimal x, decimal y);
```

If the resulting value is too large to represent in the `decimal` format, a `System.OverflowException` is thrown. If the result value is too small to represent in the `decimal` format, the result is zero. The scale of the result, before any rounding, is the sum of the scales of the two operands.

Decimal multiplication is equivalent to using the multiplication operator of type `System.Decimal`.

Division operator

For an operation of the form `x / y`, binary operator overload resolution ([Binary operator overload resolution](#)) is applied to select a specific operator implementation. The operands are converted to the parameter types of the selected operator, and the type of the result is the return type of the operator.

The predefined division operators are listed below. The operators all compute the quotient of `x` and `y`.

- Integer division:

```
int operator /(int x, int y);
uint operator /(uint x, uint y);
long operator /(long x, long y);
ulong operator /(ulong x, ulong y);
```

If the value of the right operand is zero, a `System.DivideByZeroException` is thrown.

The division rounds the result towards zero. Thus the absolute value of the result is the largest possible integer that is less than or equal to the absolute value of the quotient of the two operands. The result is zero or positive when the two operands have the same sign and zero or negative when the two operands have opposite signs.

If the left operand is the smallest representable `int` or `long` value and the right operand is `-1`, an overflow occurs. In a `checked` context, this causes a `System.ArithmeticException` (or a subclass thereof) to be thrown. In an `unchecked` context, it is implementation-defined as to whether a `System.ArithmeticException` (or a subclass thereof) is thrown or the overflow goes unreported with the resulting value being that of the left operand.

- Floating-point division:

```
float operator /(float x, float y);
double operator /(double x, double y);
```

The quotient is computed according to the rules of IEEE 754 arithmetic. The following table lists the results of all possible combinations of nonzero finite values, zeros, infinities, and NaN's. In the table, `x` and `y` are positive finite values. `z` is the result of `x / y`. If the result is too large for the destination type, `z` is infinity. If the result is too small for the destination type, `z` is zero.

	+Y	-Y	+0	-0	+INF	-INF	NAN
+x	+z	-z	+inf	-inf	+0	-0	NaN
-x	-z	+z	-inf	+inf	-0	+0	NaN
+0	+0	-0	NaN	NaN	+0	-0	NaN
-0	-0	+0	NaN	NaN	-0	+0	NaN
+inf	+inf	-inf	+inf	-inf	NaN	NaN	NaN
-inf	-inf	+inf	-inf	+inf	NaN	NaN	NaN
NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN

- Decimal division:

```
decimal operator /(decimal x, decimal y);
```

If the value of the right operand is zero, a `System.DivideByZeroException` is thrown. If the resulting value is too large to represent in the `decimal` format, a `System.OverflowException` is thrown. If the result value is too small to represent in the `decimal` format, the result is zero. The scale of the result is the smallest scale that will preserve a result equal to the nearest representable decimal value to the true mathematical result.

Decimal division is equivalent to using the division operator of type `System.Decimal`.

Remainder operator

For an operation of the form `x % y`, binary operator overload resolution ([Binary operator overload resolution](#)) is applied to select a specific operator implementation. The operands are converted to the parameter types of the selected operator, and the type of the result is the return type of the operator.

The predefined remainder operators are listed below. The operators all compute the remainder of the division between `x` and `y`.

- Integer remainder:

```
int operator %(int x, int y);
uint operator %(uint x, uint y);
long operator %(long x, long y);
ulong operator %(ulong x, ulong y);
```

The result of `x % y` is the value produced by `x - (x / y) * y`. If `y` is zero, a `System.DivideByZeroException` is thrown.

If the left operand is the smallest `int` or `long` value and the right operand is `-1`, a `System.OverflowException` is thrown. In no case does `x % y` throw an exception where `x / y` would not throw an exception.

- Floating-point remainder:

```
float operator %(float x, float y);
double operator %(double x, double y);
```

The following table lists the results of all possible combinations of nonzero finite values, zeros, infinities, and NaN's. In the table, `x` and `y` are positive finite values. `z` is the result of `x % y` and is computed as `x - n * y`, where `n` is the largest possible integer that is less than or equal to `x / y`. This method of computing the remainder is analogous to that used for integer operands, but differs from the IEEE 754 definition (in which `n` is the integer closest to `x / y`).

	+Y	-Y	+0	-0	+INF	-INF	NaN
+x	+z	+z	NaN	NaN	x	x	NaN
-x	-z	-z	NaN	NaN	-x	-x	NaN
+0	+0	+0	NaN	NaN	+0	+0	NaN
-0	-0	-0	NaN	NaN	-0	-0	NaN
+inf	NaN	NaN	NaN	NaN	NaN	NaN	NaN
-inf	NaN	NaN	NaN	NaN	NaN	NaN	NaN
NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN

- Decimal remainder:


```
decimal operator %(decimal x, decimal y);
```

If the value of the right operand is zero, a `System.DivideByZeroException` is thrown. The scale of the result, before any rounding, is the larger of the scales of the two operands, and the sign of the result, if non-zero, is the same as that of `x`.

Decimal remainder is equivalent to using the remainder operator of type `System.Decimal`.

Addition operator

For an operation of the form `x + y`, binary operator overload resolution ([Binary operator overload resolution](#)) is applied to select a specific operator implementation. The operands are converted to the parameter types of the selected operator, and the type of the result is the return type of the operator.

The predefined addition operators are listed below. For numeric and enumeration types, the predefined addition operators compute the sum of the two operands. When one or both operands are of type string, the predefined addition operators concatenate the string representation of the operands.

- Integer addition:

```
int operator +(int x, int y);
uint operator +(uint x, uint y);
long operator +(long x, long y);
ulong operator +(ulong x, ulong y);
```

In a `checked` context, if the sum is outside the range of the result type, a `System.OverflowException` is thrown. In an `unchecked` context, overflows are not reported and any significant high-order bits outside the range of the result type are discarded.

- Floating-point addition:

```
float operator +(float x, float y);
double operator +(double x, double y);
```

The sum is computed according to the rules of IEEE 754 arithmetic. The following table lists the results of all possible combinations of nonzero finite values, zeros, infinities, and NaN's. In the table, `x` and `y` are nonzero finite values, and `z` is the result of `x + y`. If `x` and `y` have the same magnitude but opposite signs, `z` is positive zero. If `x + y` is too large to represent in the destination type, `z` is an infinity with the same sign as `x + y`.

	Y	+0	-0	+INF	-INF	NaN
x	z	x	x	+inf	-inf	NaN
+0	y	+0	+0	+inf	-inf	NaN
-0	y	+0	-0	+inf	-inf	NaN
+inf	+inf	+inf	+inf	+inf	NaN	NaN
-inf	-inf	-inf	-inf	NaN	-inf	NaN
NaN	NaN	NaN	NaN	NaN	NaN	NaN

- Decimal addition:

```
decimal operator +(decimal x, decimal y);
```

If the resulting value is too large to represent in the `decimal` format, a `System.OverflowException` is thrown. The scale of the result, before any rounding, is the larger of the scales of the two operands.

Decimal addition is equivalent to using the addition operator of type `System.Decimal`.

- Enumeration addition. Every enumeration type implicitly provides the following predefined operators, where `E` is the enum type, and `U` is the underlying type of `E`:

```
E operator +(E x, U y);
E operator +(U x, E y);
```

At run-time these operators are evaluated exactly as `(E)((U)x + (U)y)`.

- String concatenation:

```
string operator +(string x, string y);
string operator +(string x, object y);
string operator +(object x, string y);
```

These overloads of the binary `+` operator perform string concatenation. If an operand of string concatenation is `null`, an empty string is substituted. Otherwise, any non-string argument is converted to its string representation by invoking the virtual `ToString` method inherited from type `object`. If `ToString` returns `null`, an empty string is substituted.

```
using System;

class Test
{
    static void Main() {
        string s = null;
        Console.WriteLine("s = >" + s + "<");           // displays s = ><
        int i = 1;
        Console.WriteLine("i = " + i);                 // displays i = 1
        float f = 1.2300E+15F;
        Console.WriteLine("f = " + f);                 // displays f = 1.23E+15
        decimal d = 2.900m;
        Console.WriteLine("d = " + d);                 // displays d = 2.900
    }
}
```

The result of the string concatenation operator is a string that consists of the characters of the left operand followed by the characters of the right operand. The string concatenation operator never returns a `null` value. A `System.OutOfMemoryException` may be thrown if there is not enough memory available to allocate the resulting string.

- Delegate combination. Every delegate type implicitly provides the following predefined operator, where `D` is the delegate type:

```
D operator +(D x, D y);
```

The binary `+` operator performs delegate combination when both operands are of some delegate type

`D`. (If the operands have different delegate types, a binding-time error occurs.) If the first operand is `null`, the result of the operation is the value of the second operand (even if that is also `null`). Otherwise, if the second operand is `null`, then the result of the operation is the value of the first operand. Otherwise, the result of the operation is a new delegate instance that, when invoked, invokes the first operand and then invokes the second operand. For examples of delegate combination, see [Subtraction operator](#) and [Delegate invocation](#). Since `System.Delegate` is not a delegate type, `operator +` is not defined for it.

Subtraction operator

For an operation of the form `x - y`, binary operator overload resolution ([Binary operator overload resolution](#)) is applied to select a specific operator implementation. The operands are converted to the parameter types of the selected operator, and the type of the result is the return type of the operator.

The predefined subtraction operators are listed below. The operators all subtract `y` from `x`.

- Integer subtraction:

```
int operator -(int x, int y);
uint operator -(uint x, uint y);
long operator -(long x, long y);
ulong operator -(ulong x, ulong y);
```

In a `checked` context, if the difference is outside the range of the result type, a `System.OverflowException` is thrown. In an `unchecked` context, overflows are not reported and any significant high-order bits outside the range of the result type are discarded.

- Floating-point subtraction:

```
float operator -(float x, float y);
double operator -(double x, double y);
```

The difference is computed according to the rules of IEEE 754 arithmetic. The following table lists the results of all possible combinations of nonzero finite values, zeros, infinities, and NaNs. In the table, `x` and `y` are nonzero finite values, and `z` is the result of `x - y`. If `x` and `y` are equal, `z` is positive zero. If `x - y` is too large to represent in the destination type, `z` is an infinity with the same sign as `x - y`.

	Y	+0	-0	+INF	-INF	NAN
x	z	x	x	-inf	+inf	NaN
+0	-y	+0	+0	-inf	+inf	NaN
-0	-y	-0	+0	-inf	+inf	NaN
+inf	+inf	+inf	+inf	NaN	+inf	NaN
-inf	-inf	-inf	-inf	-inf	NaN	NaN
NaN	NaN	NaN	NaN	NaN	NaN	NaN

- Decimal subtraction:

```
decimal operator -(decimal x, decimal y);
```

If the resulting value is too large to represent in the `decimal` format, a `System.OverflowException` is thrown. The scale of the result, before any rounding, is the larger of the scales of the two operands.

Decimal subtraction is equivalent to using the subtraction operator of type `System.Decimal`.

- Enumeration subtraction. Every enumeration type implicitly provides the following predefined operator, where `E` is the enum type, and `U` is the underlying type of `E`:

```
U operator -(E x, E y);
```

This operator is evaluated exactly as $(U)((U)x - (U)y)$. In other words, the operator computes the difference between the ordinal values of `x` and `y`, and the type of the result is the underlying type of the enumeration.

```
E operator -(E x, U y);
```

This operator is evaluated exactly as $(E)((U)x - y)$. In other words, the operator subtracts a value from the underlying type of the enumeration, yielding a value of the enumeration.

- Delegate removal. Every delegate type implicitly provides the following predefined operator, where `D` is the delegate type:

```
D operator -(D x, D y);
```

The binary `-` operator performs delegate removal when both operands are of some delegate type `D`. If the operands have different delegate types, a binding-time error occurs. If the first operand is `null`, the result of the operation is `null`. Otherwise, if the second operand is `null`, then the result of the operation is the value of the first operand. Otherwise, both operands represent invocation lists ([Delegate declarations](#)) having one or more entries, and the result is a new invocation list consisting of the first operand's list with the second operand's entries removed from it, provided the second operand's list is a proper contiguous sublist of the first's. (To determine sublist equality, corresponding entries are compared as for the delegate equality operator ([Delegate equality operators](#)).) Otherwise, the result is the value of the left operand. Neither of the operands' lists is changed in the process. If the second operand's list matches multiple sublists of contiguous entries in the first operand's list, the right-most matching sublist of contiguous entries is removed. If removal results in an empty list, the result is `null`. For example:

```

delegate void D(int x);

class C
{
    public static void M1(int i) { /* ... */ }
    public static void M2(int i) { /* ... */ }
}

class Test
{
    static void Main() {
        D cd1 = new D(C.M1);
        D cd2 = new D(C.M2);
        D cd3 = cd1 + cd2 + cd2 + cd1;    // M1 + M2 + M2 + M1
        cd3 -= cd1;                       // => M1 + M2 + M2

        cd3 = cd1 + cd2 + cd2 + cd1;      // M1 + M2 + M2 + M1
        cd3 -= cd1 + cd2;                 // => M2 + M1

        cd3 = cd1 + cd2 + cd2 + cd1;      // M1 + M2 + M2 + M1
        cd3 -= cd2 + cd2;                 // => M1 + M1

        cd3 = cd1 + cd2 + cd2 + cd1;      // M1 + M2 + M2 + M1
        cd3 -= cd2 + cd1;                 // => M1 + M2

        cd3 = cd1 + cd2 + cd2 + cd1;      // M1 + M2 + M2 + M1
        cd3 -= cd1 + cd1;                 // => M1 + M2 + M2 + M1
    }
}

```

Shift operators

The `<<` and `>>` operators are used to perform bit shifting operations.

```

shift_expression
: additive_expression
| shift_expression '<<' additive_expression
| shift_expression right_shift additive_expression
;

```

If an operand of a *shift_expression* has the compile-time type `dynamic`, then the expression is dynamically bound ([Dynamic binding](#)). In this case the compile-time type of the expression is `dynamic`, and the resolution described below will take place at run-time using the run-time type of those operands that have the compile-time type `dynamic`.

For an operation of the form `x << count` or `x >> count`, binary operator overload resolution ([Binary operator overload resolution](#)) is applied to select a specific operator implementation. The operands are converted to the parameter types of the selected operator, and the type of the result is the return type of the operator.

When declaring an overloaded shift operator, the type of the first operand must always be the class or struct containing the operator declaration, and the type of the second operand must always be `int`.

The predefined shift operators are listed below.

- Shift left:

```
int operator <<(int x, int count);
uint operator <<(uint x, int count);
long operator <<(long x, int count);
ulong operator <<(ulong x, int count);
```

The `<<` operator shifts `x` left by a number of bits computed as described below.

The high-order bits outside the range of the result type of `x` are discarded, the remaining bits are shifted left, and the low-order empty bit positions are set to zero.

- Shift right:

```
int operator >>(int x, int count);
uint operator >>(uint x, int count);
long operator >>(long x, int count);
ulong operator >>(ulong x, int count);
```

The `>>` operator shifts `x` right by a number of bits computed as described below.

When `x` is of type `int` or `long`, the low-order bits of `x` are discarded, the remaining bits are shifted right, and the high-order empty bit positions are set to zero if `x` is non-negative and set to one if `x` is negative.

When `x` is of type `uint` or `ulong`, the low-order bits of `x` are discarded, the remaining bits are shifted right, and the high-order empty bit positions are set to zero.

For the predefined operators, the number of bits to shift is computed as follows:

- When the type of `x` is `int` or `uint`, the shift count is given by the low-order five bits of `count`. In other words, the shift count is computed from `count & 0x1F`.
- When the type of `x` is `long` or `ulong`, the shift count is given by the low-order six bits of `count`. In other words, the shift count is computed from `count & 0x3F`.

If the resulting shift count is zero, the shift operators simply return the value of `x`.

Shift operations never cause overflows and produce the same results in `checked` and `unchecked` contexts.

When the left operand of the `>>` operator is of a signed integral type, the operator performs an arithmetic shift right wherein the value of the most significant bit (the sign bit) of the operand is propagated to the high-order empty bit positions. When the left operand of the `>>` operator is of an unsigned integral type, the operator performs a logical shift right wherein high-order empty bit positions are always set to zero. To perform the opposite operation of that inferred from the operand type, explicit casts can be used. For example, if `x` is a variable of type `int`, the operation `unchecked((int)((uint)x >> y))` performs a logical shift right of `x`.

Relational and type-testing operators

The `==`, `!=`, `<`, `>`, `<=`, `>=`, `is` and `as` operators are called the relational and type-testing operators.

```

relational_expression
: shift_expression
| relational_expression '<' shift_expression
| relational_expression '>' shift_expression
| relational_expression '<=' shift_expression
| relational_expression '>=' shift_expression
| relational_expression 'is' type
| relational_expression 'as' type
;

equality_expression
: relational_expression
| equality_expression '==' relational_expression
| equality_expression '!=' relational_expression
;

```

The `is` operator is described in [The is operator](#) and the `as` operator is described in [The as operator](#).

The `==`, `!=`, `<`, `>`, `<=` and `>=` operators are *comparison operators*.

If an operand of a comparison operator has the compile-time type `dynamic`, then the expression is dynamically bound ([Dynamic binding](#)). In this case the compile-time type of the expression is `dynamic`, and the resolution described below will take place at run-time using the run-time type of those operands that have the compile-time type `dynamic`.

For an operation of the form `x op y`, where `op` is a comparison operator, overload resolution ([Binary operator overload resolution](#)) is applied to select a specific operator implementation. The operands are converted to the parameter types of the selected operator, and the type of the result is the return type of the operator.

The predefined comparison operators are described in the following sections. All predefined comparison operators return a result of type `bool`, as described in the following table.

OPERATION	RESULT
<code>x == y</code>	<code>true</code> if <code>x</code> is equal to <code>y</code> , <code>false</code> otherwise
<code>x != y</code>	<code>true</code> if <code>x</code> is not equal to <code>y</code> , <code>false</code> otherwise
<code>x < y</code>	<code>true</code> if <code>x</code> is less than <code>y</code> , <code>false</code> otherwise
<code>x > y</code>	<code>true</code> if <code>x</code> is greater than <code>y</code> , <code>false</code> otherwise
<code>x <= y</code>	<code>true</code> if <code>x</code> is less than or equal to <code>y</code> , <code>false</code> otherwise
<code>x >= y</code>	<code>true</code> if <code>x</code> is greater than or equal to <code>y</code> , <code>false</code> otherwise

Integer comparison operators

The predefined integer comparison operators are:

```

bool operator ==(int x, int y);
bool operator ==(uint x, uint y);
bool operator ==(long x, long y);
bool operator ==(ulong x, ulong y);

bool operator !=(int x, int y);
bool operator !=(uint x, uint y);
bool operator !=(long x, long y);
bool operator !=(ulong x, ulong y);

bool operator <(int x, int y);
bool operator <(uint x, uint y);
bool operator <(long x, long y);
bool operator <(ulong x, ulong y);

bool operator >(int x, int y);
bool operator >(uint x, uint y);
bool operator >(long x, long y);
bool operator >(ulong x, ulong y);

bool operator <=(int x, int y);
bool operator <=(uint x, uint y);
bool operator <=(long x, long y);
bool operator <=(ulong x, ulong y);

bool operator >=(int x, int y);
bool operator >=(uint x, uint y);
bool operator >=(long x, long y);
bool operator >=(ulong x, ulong y);

```

Each of these operators compares the numeric values of the two integer operands and returns a `bool` value that indicates whether the particular relation is `true` or `false`.

Floating-point comparison operators

The predefined floating-point comparison operators are:

```

bool operator ==(float x, float y);
bool operator ==(double x, double y);

bool operator !=(float x, float y);
bool operator !=(double x, double y);

bool operator <(float x, float y);
bool operator <(double x, double y);

bool operator >(float x, float y);
bool operator >(double x, double y);

bool operator <=(float x, float y);
bool operator <=(double x, double y);

bool operator >=(float x, float y);
bool operator >=(double x, double y);

```

The operators compare the operands according to the rules of the IEEE 754 standard:

- If either operand is NaN, the result is `false` for all operators except `!=`, for which the result is `true`. For any two operands, `x != y` always produces the same result as `!(x == y)`. However, when one or both operands are NaN, the `<`, `>`, `<=`, and `>=` operators do not produce the same results as the logical negation of the opposite operator. For example, if either of `x` and `y` is NaN, then `x < y` is `false`, but `!(x >= y)` is `true`.

- When neither operand is NaN, the operators compare the values of the two floating-point operands with respect to the ordering

```
-inf < -max < ... < -min < -0.0 == +0.0 < +min < ... < +max < +inf
```

where `min` and `max` are the smallest and largest positive finite values that can be represented in the given floating-point format. Notable effects of this ordering are:

- Negative and positive zeros are considered equal.
- A negative infinity is considered less than all other values, but equal to another negative infinity.
- A positive infinity is considered greater than all other values, but equal to another positive infinity.

Decimal comparison operators

The predefined decimal comparison operators are:

```
bool operator ==(decimal x, decimal y);
bool operator !=(decimal x, decimal y);
bool operator <(decimal x, decimal y);
bool operator >(decimal x, decimal y);
bool operator <=(decimal x, decimal y);
bool operator >=(decimal x, decimal y);
```

Each of these operators compares the numeric values of the two decimal operands and returns a `bool` value that indicates whether the particular relation is `true` or `false`. Each decimal comparison is equivalent to using the corresponding relational or equality operator of type `System.Decimal`.

Boolean equality operators

The predefined boolean equality operators are:

```
bool operator ==(bool x, bool y);
bool operator !=(bool x, bool y);
```

The result of `==` is `true` if both `x` and `y` are `true` or if both `x` and `y` are `false`. Otherwise, the result is `false`.

The result of `!=` is `false` if both `x` and `y` are `true` or if both `x` and `y` are `false`. Otherwise, the result is `true`. When the operands are of type `bool`, the `!=` operator produces the same result as the `^` operator.

Enumeration comparison operators

Every enumeration type implicitly provides the following predefined comparison operators:

```
bool operator ==(E x, E y);
bool operator !=(E x, E y);
bool operator <(E x, E y);
bool operator >(E x, E y);
bool operator <=(E x, E y);
bool operator >=(E x, E y);
```

The result of evaluating `x op y`, where `x` and `y` are expressions of an enumeration type `E` with an underlying type `U`, and `op` is one of the comparison operators, is exactly the same as evaluating `((U)x) op ((U)y)`. In other words, the enumeration type comparison operators simply compare the underlying integral values of the two operands.

Reference type equality operators

The predefined reference type equality operators are:

```
bool operator ==(object x, object y);
bool operator !=(object x, object y);
```

The operators return the result of comparing the two references for equality or non-equality.

Since the predefined reference type equality operators accept operands of type `object`, they apply to all types that do not declare applicable `operator ==` and `operator !=` members. Conversely, any applicable user-defined equality operators effectively hide the predefined reference type equality operators.

The predefined reference type equality operators require one of the following:

- Both operands are a value of a type known to be a *reference_type* or the literal `null`. Furthermore, an explicit reference conversion ([Explicit reference conversions](#)) exists from the type of either operand to the type of the other operand.
- One operand is a value of type `T` where `T` is a *type_parameter* and the other operand is the literal `null`. Furthermore `T` does not have the value type constraint.

Unless one of these conditions are true, a binding-time error occurs. Notable implications of these rules are:

- It is a binding-time error to use the predefined reference type equality operators to compare two references that are known to be different at binding-time. For example, if the binding-time types of the operands are two class types `A` and `B`, and if neither `A` nor `B` derives from the other, then it would be impossible for the two operands to reference the same object. Thus, the operation is considered a binding-time error.
- The predefined reference type equality operators do not permit value type operands to be compared. Therefore, unless a struct type declares its own equality operators, it is not possible to compare values of that struct type.
- The predefined reference type equality operators never cause boxing operations to occur for their operands. It would be meaningless to perform such boxing operations, since references to the newly allocated boxed instances would necessarily differ from all other references.
- If an operand of a type parameter type `T` is compared to `null`, and the run-time type of `T` is a value type, the result of the comparison is `false`.

The following example checks whether an argument of an unconstrained type parameter type is `null`.

```
class C<T>
{
    void F(T x) {
        if (x == null) throw new ArgumentNullException();
        ...
    }
}
```

The `x == null` construct is permitted even though `T` could represent a value type, and the result is simply defined to be `false` when `T` is a value type.

For an operation of the form `x == y` or `x != y`, if any applicable `operator ==` or `operator !=` exists, the operator overload resolution ([Binary operator overload resolution](#)) rules will select that operator instead of the predefined reference type equality operator. However, it is always possible to select the predefined reference type equality operator by explicitly casting one or both of the operands to type `object`. The example

```
using System;

class Test
{
    static void Main() {
        string s = "Test";
        string t = string.Copy(s);
        Console.WriteLine(s == t);
        Console.WriteLine((object)s == t);
        Console.WriteLine(s == (object)t);
        Console.WriteLine((object)s == (object)t);
    }
}
```

produces the output

```
True
False
False
False
```

The `s` and `t` variables refer to two distinct `string` instances containing the same characters. The first comparison outputs `True` because the predefined string equality operator ([String equality operators](#)) is selected when both operands are of type `string`. The remaining comparisons all output `False` because the predefined reference type equality operator is selected when one or both of the operands are of type `object`.

Note that the above technique is not meaningful for value types. The example

```
class Test
{
    static void Main() {
        int i = 123;
        int j = 123;
        System.Console.WriteLine((object)i == (object)j);
    }
}
```

outputs `False` because the casts create references to two separate instances of boxed `int` values.

String equality operators

The predefined string equality operators are:

```
bool operator ==(string x, string y);
bool operator !=(string x, string y);
```

Two `string` values are considered equal when one of the following is true:

- Both values are `null`.
- Both values are non-null references to string instances that have identical lengths and identical characters in each character position.

The string equality operators compare string values rather than string references. When two separate string instances contain the exact same sequence of characters, the values of the strings are equal, but the references are different. As described in [Reference type equality operators](#), the reference type equality operators can be used to compare string references instead of string values.

Delegate equality operators

Every delegate type implicitly provides the following predefined comparison operators:

```
bool operator ==(System.Delegate x, System.Delegate y);
bool operator !=(System.Delegate x, System.Delegate y);
```

Two delegate instances are considered equal as follows:

- If either of the delegate instances is `null`, they are equal if and only if both are `null`.
- If the delegates have different run-time type they are never equal.
- If both of the delegate instances have an invocation list ([Delegate declarations](#)), those instances are equal if and only if their invocation lists are the same length, and each entry in one's invocation list is equal (as defined below) to the corresponding entry, in order, in the other's invocation list.

The following rules govern the equality of invocation list entries:

- If two invocation list entries both refer to the same static method then the entries are equal.
- If two invocation list entries both refer to the same non-static method on the same target object (as defined by the reference equality operators) then the entries are equal.
- Invocation list entries produced from evaluation of semantically identical *anonymous_method_expressions* or *lambda_expressions* with the same (possibly empty) set of captured outer variable instances are permitted (but not required) to be equal.

Equality operators and null

The `==` and `!=` operators permit one operand to be a value of a nullable type and the other to be the `null` literal, even if no predefined or user-defined operator (in unlifted or lifted form) exists for the operation.

For an operation of one of the forms

```
x == null
null == x
x != null
null != x
```

where `x` is an expression of a nullable type, if operator overload resolution ([Binary operator overload resolution](#)) fails to find an applicable operator, the result is instead computed from the `HasValue` property of `x`. Specifically, the first two forms are translated into `!x.HasValue`, and last two forms are translated into `x.HasValue`.

The is operator

The `is` operator is used to dynamically check if the run-time type of an object is compatible with a given type. The result of the operation `E is T`, where `E` is an expression and `T` is a type, is a boolean value indicating whether `E` can successfully be converted to type `T` by a reference conversion, a boxing conversion, or an unboxing conversion. The operation is evaluated as follows, after type arguments have been substituted for all type parameters:

- If `E` is an anonymous function, a compile-time error occurs
- If `E` is a method group or the `null` literal, or if the type of `E` is a reference type or a nullable type and the value of `E` is null, the result is false.
- Otherwise, let `D` represent the dynamic type of `E` as follows:
 - If the type of `E` is a reference type, `D` is the run-time type of the instance reference by `E`.
 - If the type of `E` is a nullable type, `D` is the underlying type of that nullable type.
 - If the type of `E` is a non-nullable value type, `D` is the type of `E`.
- The result of the operation depends on `D` and `T` as follows:

- If `T` is a reference type, the result is true if `D` and `T` are the same type, if `D` is a reference type and an implicit reference conversion from `D` to `T` exists, or if `D` is a value type and a boxing conversion from `D` to `T` exists.
- If `T` is a nullable type, the result is true if `D` is the underlying type of `T`.
- If `T` is a non-nullable value type, the result is true if `D` and `T` are the same type.
- Otherwise, the result is false.

Note that user defined conversions, are not considered by the `is` operator.

The `as` operator

The `as` operator is used to explicitly convert a value to a given reference type or nullable type. Unlike a cast expression ([Cast expressions](#)), the `as` operator never throws an exception. Instead, if the indicated conversion is not possible, the resulting value is `null`.

In an operation of the form `E as T`, `E` must be an expression and `T` must be a reference type, a type parameter known to be a reference type, or a nullable type. Furthermore, at least one of the following must be true, or otherwise a compile-time error occurs:

- An identity ([Identity conversion](#)), implicit nullable ([Implicit nullable conversions](#)), implicit reference ([Implicit reference conversions](#)), boxing ([Boxing conversions](#)), explicit nullable ([Explicit nullable conversions](#)), explicit reference ([Explicit reference conversions](#)), or unboxing ([Unboxing conversions](#)) conversion exists from `E` to `T`.
- The type of `E` or `T` is an open type.
- `E` is the `null` literal.

If the compile-time type of `E` is not `dynamic`, the operation `E as T` produces the same result as

```
E is T ? (T)(E) : (T)null
```

except that `E` is only evaluated once. The compiler can be expected to optimize `E as T` to perform at most one dynamic type check as opposed to the two dynamic type checks implied by the expansion above.

If the compile-time type of `E` is `dynamic`, unlike the cast operator the `as` operator is not dynamically bound ([Dynamic binding](#)). Therefore the expansion in this case is:

```
E is T ? (T)(object)(E) : (T)null
```

Note that some conversions, such as user defined conversions, are not possible with the `as` operator and should instead be performed using cast expressions.

In the example

```

class X
{
    public string F(object o) {
        return o as string;          // OK, string is a reference type
    }

    public T G<T>(object o) where T: Attribute {
        return o as T;               // Ok, T has a class constraint
    }

    public U H<U>(object o) {
        return o as U;               // Error, U is unconstrained
    }
}

```

the type parameter `T` of `G` is known to be a reference type, because it has the class constraint. The type parameter `U` of `H` is not however; hence the use of the `as` operator in `H` is disallowed.

Logical operators

The `&`, `^`, and `|` operators are called the logical operators.

```

and_expression
: equality_expression
| and_expression '&' equality_expression
;

exclusive_or_expression
: and_expression
| exclusive_or_expression '^' and_expression
;

inclusive_or_expression
: exclusive_or_expression
| inclusive_or_expression '|' exclusive_or_expression
;

```

If an operand of a logical operator has the compile-time type `dynamic`, then the expression is dynamically bound ([Dynamic binding](#)). In this case the compile-time type of the expression is `dynamic`, and the resolution described below will take place at run-time using the run-time type of those operands that have the compile-time type `dynamic`.

For an operation of the form `x op y`, where `op` is one of the logical operators, overload resolution ([Binary operator overload resolution](#)) is applied to select a specific operator implementation. The operands are converted to the parameter types of the selected operator, and the type of the result is the return type of the operator.

The predefined logical operators are described in the following sections.

Integer logical operators

The predefined integer logical operators are:

```

int operator &(int x, int y);
uint operator &(uint x, uint y);
long operator &(long x, long y);
ulong operator &(ulong x, ulong y);

int operator |(int x, int y);
uint operator |(uint x, uint y);
long operator |(long x, long y);
ulong operator |(ulong x, ulong y);

int operator ^(int x, int y);
uint operator ^(uint x, uint y);
long operator ^(long x, long y);
ulong operator ^(ulong x, ulong y);

```

The `&` operator computes the bitwise logical `AND` of the two operands, the `|` operator computes the bitwise logical `OR` of the two operands, and the `^` operator computes the bitwise logical exclusive `OR` of the two operands. No overflows are possible from these operations.

Enumeration logical operators

Every enumeration type `E` implicitly provides the following predefined logical operators:

```

E operator &(E x, E y);
E operator |(E x, E y);
E operator ^(E x, E y);

```

The result of evaluating `x op y`, where `x` and `y` are expressions of an enumeration type `E` with an underlying type `U`, and `op` is one of the logical operators, is exactly the same as evaluating `(E)((U)x op (U)y)`. In other words, the enumeration type logical operators simply perform the logical operation on the underlying type of the two operands.

Boolean logical operators

The predefined boolean logical operators are:

```

bool operator &(bool x, bool y);
bool operator |(bool x, bool y);
bool operator ^(bool x, bool y);

```

The result of `x & y` is `true` if both `x` and `y` are `true`. Otherwise, the result is `false`.

The result of `x | y` is `true` if either `x` or `y` is `true`. Otherwise, the result is `false`.

The result of `x ^ y` is `true` if `x` is `true` and `y` is `false`, or `x` is `false` and `y` is `true`. Otherwise, the result is `false`. When the operands are of type `bool`, the `^` operator computes the same result as the `!=` operator.

Nullable boolean logical operators

The nullable boolean type `bool?` can represent three values, `true`, `false`, and `null`, and is conceptually similar to the three-valued type used for boolean expressions in SQL. To ensure that the results produced by the `&` and `|` operators for `bool?` operands are consistent with SQL's three-valued logic, the following predefined operators are provided:

```

bool? operator &(bool? x, bool? y);
bool? operator |(bool? x, bool? y);

```

The following table lists the results produced by these operators for all combinations of the values `true`, `false`, and `null`.

x	y	x & y	x y
true	true	true	true
true	false	false	true
true	null	null	true
false	true	false	true
false	false	false	false
false	null	false	null
null	true	null	true
null	false	false	null
null	null	null	null

Conditional logical operators

The `&&` and `||` operators are called the conditional logical operators. They are also called the "short-circuiting" logical operators.

```
conditional_and_expression
: inclusive_or_expression
| conditional_and_expression '&&' inclusive_or_expression
;

conditional_or_expression
: conditional_and_expression
| conditional_or_expression '||' conditional_and_expression
;
```

The `&&` and `||` operators are conditional versions of the `&` and `|` operators:

- The operation `x && y` corresponds to the operation `x & y`, except that `y` is evaluated only if `x` is not `false`.
- The operation `x || y` corresponds to the operation `x | y`, except that `y` is evaluated only if `x` is not `true`.

If an operand of a conditional logical operator has the compile-time type `dynamic`, then the expression is dynamically bound ([Dynamic binding](#)). In this case the compile-time type of the expression is `dynamic`, and the resolution described below will take place at run-time using the run-time type of those operands that have the compile-time type `dynamic`.

An operation of the form `x && y` or `x || y` is processed by applying overload resolution ([Binary operator overload resolution](#)) as if the operation was written `x & y` or `x | y`. Then,

- If overload resolution fails to find a single best operator, or if overload resolution selects one of the

predefined integer logical operators, a binding-time error occurs.

- Otherwise, if the selected operator is one of the predefined boolean logical operators ([Boolean logical operators](#)) or nullable boolean logical operators ([Nullable boolean logical operators](#)), the operation is processed as described in [Boolean conditional logical operators](#).
- Otherwise, the selected operator is a user-defined operator, and the operation is processed as described in [User-defined conditional logical operators](#).

It is not possible to directly overload the conditional logical operators. However, because the conditional logical operators are evaluated in terms of the regular logical operators, overloads of the regular logical operators are, with certain restrictions, also considered overloads of the conditional logical operators. This is described further in [User-defined conditional logical operators](#).

Boolean conditional logical operators

When the operands of `&&` or `||` are of type `bool`, or when the operands are of types that do not define an applicable `operator &` or `operator |`, but do define implicit conversions to `bool`, the operation is processed as follows:

- The operation `x && y` is evaluated as `x ? y : false`. In other words, `x` is first evaluated and converted to type `bool`. Then, if `x` is `true`, `y` is evaluated and converted to type `bool`, and this becomes the result of the operation. Otherwise, the result of the operation is `false`.
- The operation `x || y` is evaluated as `x ? true : y`. In other words, `x` is first evaluated and converted to type `bool`. Then, if `x` is `true`, the result of the operation is `true`. Otherwise, `y` is evaluated and converted to type `bool`, and this becomes the result of the operation.

User-defined conditional logical operators

When the operands of `&&` or `||` are of types that declare an applicable user-defined `operator &` or `operator |`, both of the following must be true, where `T` is the type in which the selected operator is declared:

- The return type and the type of each parameter of the selected operator must be `T`. In other words, the operator must compute the logical `AND` or the logical `OR` of two operands of type `T`, and must return a result of type `T`.
- `T` must contain declarations of `operator true` and `operator false`.

A binding-time error occurs if either of these requirements is not satisfied. Otherwise, the `&&` or `||` operation is evaluated by combining the user-defined `operator true` or `operator false` with the selected user-defined operator:

- The operation `x && y` is evaluated as `T.false(x) ? x : T.&(x, y)`, where `T.false(x)` is an invocation of the `operator false` declared in `T`, and `T.&(x, y)` is an invocation of the selected `operator &`. In other words, `x` is first evaluated and `operator false` is invoked on the result to determine if `x` is definitely false. Then, if `x` is definitely false, the result of the operation is the value previously computed for `x`. Otherwise, `y` is evaluated, and the selected `operator &` is invoked on the value previously computed for `x` and the value computed for `y` to produce the result of the operation.
- The operation `x || y` is evaluated as `T.true(x) ? x : T.|(x, y)`, where `T.true(x)` is an invocation of the `operator true` declared in `T`, and `T.|(x, y)` is an invocation of the selected `operator |`. In other words, `x` is first evaluated and `operator true` is invoked on the result to determine if `x` is definitely true. Then, if `x` is definitely true, the result of the operation is the value previously computed for `x`. Otherwise, `y` is evaluated, and the selected `operator |` is invoked on the value previously computed for `x` and the value computed for `y` to produce the result of the operation.

In either of these operations, the expression given by `x` is only evaluated once, and the expression given by `y` is either not evaluated or evaluated exactly once.

For an example of a type that implements `operator true` and `operator false`, see [Database boolean type](#).

The null coalescing operator

The `??` operator is called the null coalescing operator.

```
null_coalescing_expression
: conditional_or_expression
| conditional_or_expression '??' null_coalescing_expression
;
```

A null coalescing expression of the form `a ?? b` requires `a` to be of a nullable type or reference type. If `a` is non-null, the result of `a ?? b` is `a`; otherwise, the result is `b`. The operation evaluates `b` only if `a` is null.

The null coalescing operator is right-associative, meaning that operations are grouped from right to left. For example, an expression of the form `a ?? b ?? c` is evaluated as `a ?? (b ?? c)`. In general terms, an expression of the form `E1 ?? E2 ?? ... ?? En` returns the first of the operands that is non-null, or null if all operands are null.

The type of the expression `a ?? b` depends on which implicit conversions are available on the operands. In order of preference, the type of `a ?? b` is `A0`, `A`, or `B`, where `A` is the type of `a` (provided that `a` has a type), `B` is the type of `b` (provided that `b` has a type), and `A0` is the underlying type of `A` if `A` is a nullable type, or `A` otherwise. Specifically, `a ?? b` is processed as follows:

- If `A` exists and is not a nullable type or a reference type, a compile-time error occurs.
- If `b` is a dynamic expression, the result type is `dynamic`. At run-time, `a` is first evaluated. If `a` is not null, `a` is converted to dynamic, and this becomes the result. Otherwise, `b` is evaluated, and this becomes the result.
- Otherwise, if `A` exists and is a nullable type and an implicit conversion exists from `b` to `A0`, the result type is `A0`. At run-time, `a` is first evaluated. If `a` is not null, `a` is unwrapped to type `A0`, and this becomes the result. Otherwise, `b` is evaluated and converted to type `A0`, and this becomes the result.
- Otherwise, if `A` exists and an implicit conversion exists from `b` to `A`, the result type is `A`. At run-time, `a` is first evaluated. If `a` is not null, `a` becomes the result. Otherwise, `b` is evaluated and converted to type `A`, and this becomes the result.
- Otherwise, if `b` has a type `B` and an implicit conversion exists from `a` to `B`, the result type is `B`. At run-time, `a` is first evaluated. If `a` is not null, `a` is unwrapped to type `A0` (if `A` exists and is nullable) and converted to type `B`, and this becomes the result. Otherwise, `b` is evaluated and becomes the result.
- Otherwise, `a` and `b` are incompatible, and a compile-time error occurs.

Conditional operator

The `?:` operator is called the conditional operator. It is at times also called the ternary operator.

```
conditional_expression
: null_coalescing_expression
| null_coalescing_expression '?' expression ':' expression
;
```

A conditional expression of the form `b ? x : y` first evaluates the condition `b`. Then, if `b` is `true`, `x` is evaluated and becomes the result of the operation. Otherwise, `y` is evaluated and becomes the result of the operation. A conditional expression never evaluates both `x` and `y`.

The conditional operator is right-associative, meaning that operations are grouped from right to left. For

example, an expression of the form `a ? b : c ? d : e` is evaluated as `a ? b : (c ? d : e)`.

The first operand of the `?:` operator must be an expression that can be implicitly converted to `bool`, or an expression of a type that implements `operator true`. If neither of these requirements is satisfied, a compile-time error occurs.

The second and third operands, `x` and `y`, of the `?:` operator control the type of the conditional expression.

- If `x` has type `x` and `y` has type `y` then
 - If an implicit conversion ([Implicit conversions](#)) exists from `x` to `y`, but not from `y` to `x`, then `y` is the type of the conditional expression.
 - If an implicit conversion ([Implicit conversions](#)) exists from `y` to `x`, but not from `x` to `y`, then `x` is the type of the conditional expression.
 - Otherwise, no expression type can be determined, and a compile-time error occurs.
- If only one of `x` and `y` has a type, and both `x` and `y` are implicitly convertible to that type, then that is the type of the conditional expression.
- Otherwise, no expression type can be determined, and a compile-time error occurs.

The run-time processing of a conditional expression of the form `b ? x : y` consists of the following steps:

- First, `b` is evaluated, and the `bool` value of `b` is determined:
 - If an implicit conversion from the type of `b` to `bool` exists, then this implicit conversion is performed to produce a `bool` value.
 - Otherwise, the `operator true` defined by the type of `b` is invoked to produce a `bool` value.
- If the `bool` value produced by the step above is `true`, then `x` is evaluated and converted to the type of the conditional expression, and this becomes the result of the conditional expression.
- Otherwise, `y` is evaluated and converted to the type of the conditional expression, and this becomes the result of the conditional expression.

Anonymous function expressions

An **anonymous function** is an expression that represents an "in-line" method definition. An anonymous function does not have a value or type in and of itself, but is convertible to a compatible delegate or expression tree type. The evaluation of an anonymous function conversion depends on the target type of the conversion: If it is a delegate type, the conversion evaluates to a delegate value referencing the method which the anonymous function defines. If it is an expression tree type, the conversion evaluates to an expression tree which represents the structure of the method as an object structure.

For historical reasons there are two syntactic flavors of anonymous functions, namely *lambda_expressions* and *anonymous_method_expressions*. For almost all purposes, *lambda_expressions* are more concise and expressive than *anonymous_method_expressions*, which remain in the language for backwards compatibility.

```

lambda_expression
  : anonymous_function_signature '=>' anonymous_function_body
  ;

anonymous_method_expression
  : 'delegate' explicit_anonymous_function_signature? block
  ;

anonymous_function_signature
  : explicit_anonymous_function_signature
  | implicit_anonymous_function_signature
  ;

explicit_anonymous_function_signature
  : '(' explicit_anonymous_function_parameter_list? ')'
  ;

explicit_anonymous_function_parameter_list
  : explicit_anonymous_function_parameter (',' explicit_anonymous_function_parameter)*
  ;

explicit_anonymous_function_parameter
  : anonymous_function_parameter_modifier? type identifier
  ;

anonymous_function_parameter_modifier
  : 'ref'
  | 'out'
  ;

implicit_anonymous_function_signature
  : '(' implicit_anonymous_function_parameter_list? ')'
  | implicit_anonymous_function_parameter
  ;

implicit_anonymous_function_parameter_list
  : implicit_anonymous_function_parameter (',' implicit_anonymous_function_parameter)*
  ;

implicit_anonymous_function_parameter
  : identifier
  ;

anonymous_function_body
  : expression
  | block
  ;

```

The `=>` operator has the same precedence as assignment (`=`) and is right-associative.

An anonymous function with the `async` modifier is an async function and follows the rules described in [Async functions](#).

The parameters of an anonymous function in the form of a *lambda_expression* can be explicitly or implicitly typed. In an explicitly typed parameter list, the type of each parameter is explicitly stated. In an implicitly typed parameter list, the types of the parameters are inferred from the context in which the anonymous function occurs—specifically, when the anonymous function is converted to a compatible delegate type or expression tree type, that type provides the parameter types ([Anonymous function conversions](#)).

In an anonymous function with a single, implicitly typed parameter, the parentheses may be omitted from the parameter list. In other words, an anonymous function of the form

```
( param ) => expr
```

can be abbreviated to

```
param => expr
```

The parameter list of an anonymous function in the form of an *anonymous_method_expression* is optional. If given, the parameters must be explicitly typed. If not, the anonymous function is convertible to a delegate with any parameter list not containing `out` parameters.

A *block* body of an anonymous function is reachable ([End points and reachability](#)) unless the anonymous function occurs inside an unreachable statement.

Some examples of anonymous functions follow below:

```
x => x + 1 // Implicitly typed, expression body
x => { return x + 1; } // Implicitly typed, statement body
(int x) => x + 1 // Explicitly typed, expression body
(int x) => { return x + 1; } // Explicitly typed, statement body
(x, y) => x * y // Multiple parameters
() => Console.WriteLine() // No parameters
async (t1,t2) => await t1 + await t2 // Async
delegate (int x) { return x + 1; } // Anonymous method expression
delegate { return 1 + 1; } // Parameter list omitted
```

The behavior of *lambda_expressions* and *anonymous_method_expressions* is the same except for the following points:

- *anonymous_method_expressions* permit the parameter list to be omitted entirely, yielding convertibility to delegate types of any list of value parameters.
- *lambda_expressions* permit parameter types to be omitted and inferred whereas *anonymous_method_expressions* require parameter types to be explicitly stated.
- The body of a *lambda_expression* can be an expression or a statement block whereas the body of an *anonymous_method_expression* must be a statement block.
- Only *lambda_expressions* have conversions to compatible expression tree types ([Expression tree types](#)).

Anonymous function signatures

The optional *anonymous_function_signature* of an anonymous function defines the names and optionally the types of the formal parameters for the anonymous function. The scope of the parameters of the anonymous function is the *anonymous_function_body*. ([Scopes](#)) Together with the parameter list (if given) the anonymous-method-body constitutes a declaration space ([Declarations](#)). It is thus a compile-time error for the name of a parameter of the anonymous function to match the name of a local variable, local constant or parameter whose scope includes the *anonymous_method_expression* or *lambda_expression*.

If an anonymous function has an *explicit_anonymous_function_signature*, then the set of compatible delegate types and expression tree types is restricted to those that have the same parameter types and modifiers in the same order. In contrast to method group conversions ([Method group conversions](#)), contra-variance of anonymous function parameter types is not supported. If an anonymous function does not have an *anonymous_function_signature*, then the set of compatible delegate types and expression tree types is restricted to those that have no `out` parameters.

Note that an *anonymous_function_signature* cannot include attributes or a parameter array. Nevertheless, an *anonymous_function_signature* may be compatible with a delegate type whose parameter list contains a parameter array.

Note also that conversion to an expression tree type, even if compatible, may still fail at compile-time ([Expression tree types](#)).

Anonymous function bodies

The body (*expression* or *block*) of an anonymous function is subject to the following rules:

- If the anonymous function includes a signature, the parameters specified in the signature are available in the body. If the anonymous function has no signature it can be converted to a delegate type or expression type having parameters ([Anonymous function conversions](#)), but the parameters cannot be accessed in the body.
- Except for `ref` or `out` parameters specified in the signature (if any) of the nearest enclosing anonymous function, it is a compile-time error for the body to access a `ref` or `out` parameter.
- When the type of `this` is a struct type, it is a compile-time error for the body to access `this`. This is true whether the access is explicit (as in `this.x`) or implicit (as in `x` where `x` is an instance member of the struct). This rule simply prohibits such access and does not affect whether member lookup results in a member of the struct.
- The body has access to the outer variables ([Outer variables](#)) of the anonymous function. Access of an outer variable will reference the instance of the variable that is active at the time the *lambda_expression* or *anonymous_method_expression* is evaluated ([Evaluation of anonymous function expressions](#)).
- It is a compile-time error for the body to contain a `goto` statement, `break` statement, or `continue` statement whose target is outside the body or within the body of a contained anonymous function.
- A `return` statement in the body returns control from an invocation of the nearest enclosing anonymous function, not from the enclosing function member. An expression specified in a `return` statement must be implicitly convertible to the return type of the delegate type or expression tree type to which the nearest enclosing *lambda_expression* or *anonymous_method_expression* is converted ([Anonymous function conversions](#)).

It is explicitly unspecified whether there is any way to execute the block of an anonymous function other than through evaluation and invocation of the *lambda_expression* or *anonymous_method_expression*. In particular, the compiler may choose to implement an anonymous function by synthesizing one or more named methods or types. The names of any such synthesized elements must be of a form reserved for compiler use.

Overload resolution and anonymous functions

Anonymous functions in an argument list participate in type inference and overload resolution. Please refer to [Type inference](#) and [Overload resolution](#) for the exact rules.

The following example illustrates the effect of anonymous functions on overload resolution.

```
class ItemList<T>: List<T>
{
    public int Sum(Func<T,int> selector) {
        int sum = 0;
        foreach (T item in this) sum += selector(item);
        return sum;
    }

    public double Sum(Func<T,double> selector) {
        double sum = 0;
        foreach (T item in this) sum += selector(item);
        return sum;
    }
}
```

The `ItemList<T>` class has two `Sum` methods. Each takes a `selector` argument, which extracts the value to sum over from a list item. The extracted value can be either an `int` or a `double` and the resulting sum is likewise either an `int` or a `double`.

The `sum` methods could for example be used to compute sums from a list of detail lines in an order.

```
class Detail
{
    public int UnitCount;
    public double UnitPrice;
    ...
}

void ComputeSums() {
    ItemList<Detail> orderDetails = GetOrderDetails(...);
    int totalUnits = orderDetails.Sum(d => d.UnitCount);
    double orderTotal = orderDetails.Sum(d => d.UnitPrice * d.UnitCount);
    ...
}
```

In the first invocation of `orderDetails.Sum`, both `Sum` methods are applicable because the anonymous function `d => d. UnitCount` is compatible with both `Func<Detail,int>` and `Func<Detail,double>`. However, overload resolution picks the first `Sum` method because the conversion to `Func<Detail,int>` is better than the conversion to `Func<Detail,double>`.

In the second invocation of `orderDetails.Sum`, only the second `Sum` method is applicable because the anonymous function `d => d.UnitPrice * d.UnitCount` produces a value of type `double`. Thus, overload resolution picks the second `Sum` method for that invocation.

Anonymous functions and dynamic binding

An anonymous function cannot be a receiver, argument or operand of a dynamically bound operation.

Outer variables

Any local variable, value parameter, or parameter array whose scope includes the *lambda_expression* or *anonymous_method_expression* is called an **outer variable** of the anonymous function. In an instance function member of a class, the `this` value is considered a value parameter and is an outer variable of any anonymous function contained within the function member.

Captured outer variables

When an outer variable is referenced by an anonymous function, the outer variable is said to have been **captured** by the anonymous function. Ordinarily, the lifetime of a local variable is limited to execution of the block or statement with which it is associated ([Local variables](#)). However, the lifetime of a captured outer variable is extended at least until the delegate or expression tree created from the anonymous function becomes eligible for garbage collection.

In the example

```

using System;

delegate int D();

class Test
{
    static D F() {
        int x = 0;
        D result = () => ++x;
        return result;
    }

    static void Main() {
        D d = F();
        Console.WriteLine(d());
        Console.WriteLine(d());
        Console.WriteLine(d());
    }
}

```

the local variable `x` is captured by the anonymous function, and the lifetime of `x` is extended at least until the delegate returned from `F` becomes eligible for garbage collection (which doesn't happen until the very end of the program). Since each invocation of the anonymous function operates on the same instance of `x`, the output of the example is:

```

1
2
3

```

When a local variable or a value parameter is captured by an anonymous function, the local variable or parameter is no longer considered to be a fixed variable ([Fixed and moveable variables](#)), but is instead considered to be a moveable variable. Thus any `unsafe` code that takes the address of a captured outer variable must first use the `fixed` statement to fix the variable.

Note that unlike an uncaptured variable, a captured local variable can be simultaneously exposed to multiple threads of execution.

Instantiation of local variables

A local variable is considered to be *instantiated* when execution enters the scope of the variable. For example, when the following method is invoked, the local variable `x` is instantiated and initialized three times—once for each iteration of the loop.

```

static void F() {
    for (int i = 0; i < 3; i++) {
        int x = i * 2 + 1;
        ...
    }
}

```

However, moving the declaration of `x` outside the loop results in a single instantiation of `x`:


```
static void F() {
    int x;
    for (int i = 0; i < 3; i++) {
        x = i * 2 + 1;
        ...
    }
}
```

When not captured, there is no way to observe exactly how often a local variable is instantiated—because the lifetimes of the instantiations are disjoint, it is possible for each instantiation to simply use the same storage location. However, when an anonymous function captures a local variable, the effects of instantiation become apparent.

The example

```
using System;

delegate void D();

class Test
{
    static D[] F() {
        D[] result = new D[3];
        for (int i = 0; i < 3; i++) {
            int x = i * 2 + 1;
            result[i] = () => { Console.WriteLine(x); };
        }
        return result;
    }

    static void Main() {
        foreach (D d in F()) d();
    }
}
```

produces the output:

```
1
3
5
```

However, when the declaration of `x` is moved outside the loop:

```
static D[] F() {
    D[] result = new D[3];
    int x;
    for (int i = 0; i < 3; i++) {
        x = i * 2 + 1;
        result[i] = () => { Console.WriteLine(x); };
    }
    return result;
}
```

the output is:

```
5
5
5
```

If a for-loop declares an iteration variable, that variable itself is considered to be declared outside of the loop. Thus, if the example is changed to capture the iteration variable itself:

```
static D[] F() {
    D[] result = new D[3];
    for (int i = 0; i < 3; i++) {
        result[i] = () => { Console.WriteLine(i); };
    }
    return result;
}
```

only one instance of the iteration variable is captured, which produces the output:

```
3
3
3
```

It is possible for anonymous function delegates to share some captured variables yet have separate instances of others. For example, if `F` is changed to

```
static D[] F() {
    D[] result = new D[3];
    int x = 0;
    for (int i = 0; i < 3; i++) {
        int y = 0;
        result[i] = () => { Console.WriteLine("{0} {1}", ++x, ++y); };
    }
    return result;
}
```

the three delegates capture the same instance of `x` but separate instances of `y`, and the output is:

```
1 1
2 1
3 1
```

Separate anonymous functions can capture the same instance of an outer variable. In the example:

```
using System;

delegate void Setter(int value);

delegate int Getter();

class Test
{
    static void Main() {
        int x = 0;
        Setter s = (int value) => { x = value; };
        Getter g = () => { return x; };
        s(5);
        Console.WriteLine(g());
        s(10);
        Console.WriteLine(g());
    }
}
```

the two anonymous functions capture the same instance of the local variable `x`, and they can thus

"communicate" through that variable. The output of the example is:

```
5
10
```

Evaluation of anonymous function expressions

An anonymous function `F` must always be converted to a delegate type `D` or an expression tree type `E`, either directly or through the execution of a delegate creation expression `new D(F)`. This conversion determines the result of the anonymous function, as described in [Anonymous function conversions](#).

Query expressions

Query expressions provide a language integrated syntax for queries that is similar to relational and hierarchical query languages such as SQL and XQuery.

```
query_expression
    : from_clause query_body
    ;

from_clause
    : 'from' type? identifier 'in' expression
    ;

query_body
    : query_body_clauses? select_or_group_clause query_continuation?
    ;

query_body_clauses
    : query_body_clause
    | query_body_clauses query_body_clause
    ;

query_body_clause
    : from_clause
    | let_clause
    | where_clause
    | join_clause
    | join_into_clause
    | orderby_clause
    ;

let_clause
    : 'let' identifier '=' expression
    ;

where_clause
    : 'where' boolean_expression
    ;

join_clause
    : 'join' type? identifier 'in' expression 'on' expression 'equals' expression
    ;

join_into_clause
    : 'join' type? identifier 'in' expression 'on' expression 'equals' expression 'into' identifier
    ;

orderby_clause
    : 'orderby' orderings
    ;

orderings
    : ordering (',' ordering)*
```

```

;

ordering
: expression ordering_direction?
;

ordering_direction
: 'ascending'
| 'descending'
;

select_or_group_clause
: select_clause
| group_clause
;

select_clause
: 'select' expression
;

group_clause
: 'group' expression 'by' expression
;

query_continuation
: 'into' identifier query_body
;

```

A query expression begins with a `from` clause and ends with either a `select` or `group` clause. The initial `from` clause can be followed by zero or more `from`, `let`, `where`, `join` or `orderby` clauses. Each `from` clause is a generator introducing a **range variable** which ranges over the elements of a **sequence**. Each `let` clause introduces a range variable representing a value computed by means of previous range variables. Each `where` clause is a filter that excludes items from the result. Each `join` clause compares specified keys of the source sequence with keys of another sequence, yielding matching pairs. Each `orderby` clause reorders items according to specified criteria. The final `select` or `group` clause specifies the shape of the result in terms of the range variables. Finally, an `into` clause can be used to "splice" queries by treating the results of one query as a generator in a subsequent query.

Ambiguities in query expressions

Query expressions contain a number of "contextual keywords", i.e., identifiers that have special meaning in a given context. Specifically these are `from`, `where`, `join`, `on`, `equals`, `into`, `let`, `orderby`, `ascending`, `descending`, `select`, `group` and `by`. In order to avoid ambiguities in query expressions caused by mixed use of these identifiers as keywords or simple names, these identifiers are considered keywords when occurring anywhere within a query expression.

For this purpose, a query expression is any expression that starts with "`from identifier`" followed by any token except "`;`", "`=`" or "`,`".

In order to use these words as identifiers within a query expression, they can be prefixed with "`@`" (**Identifiers**).

Query expression translation

The C# language does not specify the execution semantics of query expressions. Rather, query expressions are translated into invocations of methods that adhere to the *query expression pattern* ([The query expression pattern](#)). Specifically, query expressions are translated into invocations of methods named `Where`, `Select`, `SelectMany`, `Join`, `GroupJoin`, `OrderBy`, `OrderByDescending`, `ThenBy`, `ThenByDescending`, `GroupBy`, and `Cast`. These methods are expected to have particular signatures and result types, as described in [The query expression pattern](#). These methods can be instance methods of the object being queried or extension methods that are external to the object, and they implement the actual execution of the query.

The translation from query expressions to method invocations is a syntactic mapping that occurs before any

type binding or overload resolution has been performed. The translation is guaranteed to be syntactically correct, but it is not guaranteed to produce semantically correct C# code. Following translation of query expressions, the resulting method invocations are processed as regular method invocations, and this may in turn uncover errors, for example if the methods do not exist, if arguments have wrong types, or if the methods are generic and type inference fails.

A query expression is processed by repeatedly applying the following translations until no further reductions are possible. The translations are listed in order of application: each section assumes that the translations in the preceding sections have been performed exhaustively, and once exhausted, a section will not later be revisited in the processing of the same query expression.

Assignment to range variables is not allowed in query expressions. However a C# implementation is permitted to not always enforce this restriction, since this may sometimes not be possible with the syntactic translation scheme presented here.

Certain translations inject range variables with transparent identifiers denoted by `*`. The special properties of transparent identifiers are discussed further in [Transparent identifiers](#).

Select and groupby clauses with continuations

A query expression with a continuation

```
from ... into x ...
```

is translated into

```
from x in ( from ... ) ...
```

The translations in the following sections assume that queries have no `into` continuations.

The example

```
from c in customers
group c by c.Country into g
select new { Country = g.Key, CustCount = g.Count() }
```

is translated into

```
from g in
    from c in customers
    group c by c.Country
select new { Country = g.Key, CustCount = g.Count() }
```

the final translation of which is

```
customers.
  GroupBy(c => c.Country).
  Select(g => new { Country = g.Key, CustCount = g.Count() })
```

Explicit range variable types

A `from` clause that explicitly specifies a range variable type

```
from T x in e
```

is translated into

```
from x in ( e ) . Cast < T > ( )
```

A `join` clause that explicitly specifies a range variable type

```
join T x in e on k1 equals k2
```

is translated into

```
join x in ( e ) . Cast < T > ( ) on k1 equals k2
```

The translations in the following sections assume that queries have no explicit range variable types.

The example

```
from Customer c in customers
where c.City == "London"
select c
```

is translated into

```
from c in customers.Cast<Customer>()
where c.City == "London"
select c
```

the final translation of which is

```
customers.
Cast<Customer>().
Where(c => c.City == "London")
```

Explicit range variable types are useful for querying collections that implement the non-generic `IEnumerable` interface, but not the generic `IEnumerable<T>` interface. In the example above, this would be the case if `customers` were of type `ArrayList`.

Degenerate query expressions

A query expression of the form

```
from x in e select x
```

is translated into

```
( e ) . Select ( x => x )
```

The example

```
from c in customers
select c
```

is translated into

```
customers.Select(c => c)
```

A degenerate query expression is one that trivially selects the elements of the source. A later phase of the translation removes degenerate queries introduced by other translation steps by replacing them with their source. It is important however to ensure that the result of a query expression is never the source object itself, as that would reveal the type and identity of the source to the client of the query. Therefore this step protects degenerate queries written directly in source code by explicitly calling `Select` on the source. It is then up to the implementers of `Select` and other query operators to ensure that these methods never return the source object itself.

From, let, where, join and orderby clauses

A query expression with a second `from` clause followed by a `select` clause

```
from x1 in e1
from x2 in e2
select v
```

is translated into

```
( e1 ). SelectMany( x1 => e2 , ( x1 , x2 ) => v )
```

A query expression with a second `from` clause followed by something other than a `select` clause:

```
from x1 in e1
from x2 in e2
...
```

is translated into

```
from * in ( e1 ). SelectMany( x1 => e2 , ( x1 , x2 ) => new { x1 , x2 } )
...
```

A query expression with a `let` clause

```
from x in e
let y = f
...
```

is translated into

```
from * in ( e ) . Select ( x => new { x , y = f } )
...
```

A query expression with a `where` clause

```
from x in e
where f
...
```

is translated into

```
from x in ( e ) . Where ( x => f )
...
```

A query expression with a `join` clause without an `into` followed by a `select` clause

```
from x1 in e1
join x2 in e2 on k1 equals k2
select v
```

is translated into

```
( e1 ) . Join( e2 , x1 => k1 , x2 => k2 , ( x1 , x2 ) => v )
```

A query expression with a `join` clause without an `into` followed by something other than a `select` clause

```
from x1 in e1
join x2 in e2 on k1 equals k2
...
```

is translated into

```
from * in ( e1 ) . Join( e2 , x1 => k1 , x2 => k2 , ( x1 , x2 ) => new { x1 , x2 })
...
```

A query expression with a `join` clause with an `into` followed by a `select` clause

```
from x1 in e1
join x2 in e2 on k1 equals k2 into g
select v
```

is translated into

```
( e1 ) . GroupJoin( e2 , x1 => k1 , x2 => k2 , ( x1 , g ) => v )
```

A query expression with a `join` clause with an `into` followed by something other than a `select` clause

```
from x1 in e1
join x2 in e2 on k1 equals k2 into g
...
```

is translated into

```
from * in ( e1 ) . GroupJoin( e2 , x1 => k1 , x2 => k2 , ( x1 , g ) => new { x1 , g })
...
```

A query expression with an `orderby` clause

```
from x in e
orderby k1 , k2 , ..., kn
...
```


is translated into

```
from x in ( e ) .  
OrderBy ( x => k1 ) .  
ThenBy ( x => k2 ) .  
... .  
ThenBy ( x => kn )  
...
```

If an ordering clause specifies a `descending` direction indicator, an invocation of `OrderByDescending` or `ThenByDescending` is produced instead.

The following translations assume that there are no `let`, `where`, `join` or `orderby` clauses, and no more than the one initial `from` clause in each query expression.

The example

```
from c in customers  
from o in c.Orders  
select new { c.Name, o.OrderID, o.Total }
```

is translated into

```
customers.  
SelectMany(c => c.Orders,  
    (c,o) => new { c.Name, o.OrderID, o.Total }  
)
```

The example

```
from c in customers  
from o in c.Orders  
orderby o.Total descending  
select new { c.Name, o.OrderID, o.Total }
```

is translated into

```
from * in customers.  
    SelectMany(c => c.Orders, (c,o) => new { c, o })  
orderby o.Total descending  
select new { c.Name, o.OrderID, o.Total }
```

the final translation of which is

```
customers.  
SelectMany(c => c.Orders, (c,o) => new { c, o }).  
OrderByDescending(x => x.o.Total).  
Select(x => new { x.c.Name, x.o.OrderID, x.o.Total })
```

where `x` is a compiler generated identifier that is otherwise invisible and inaccessible.

The example

```

from o in orders
let t = o.Details.Sum(d => d.UnitPrice * d.Quantity)
where t >= 1000
select new { o.OrderID, Total = t }

```

is translated into

```

from * in orders.
    Select(o => new { o, t = o.Details.Sum(d => d.UnitPrice * d.Quantity) })
where t >= 1000
select new { o.OrderID, Total = t }

```

the final translation of which is

```

orders.
Select(o => new { o, t = o.Details.Sum(d => d.UnitPrice * d.Quantity) }).
Where(x => x.t >= 1000).
Select(x => new { x.o.OrderID, Total = x.t })

```

where `x` is a compiler generated identifier that is otherwise invisible and inaccessible.

The example

```

from c in customers
join o in orders on c.CustomerID equals o.CustomerID
select new { c.Name, o.OrderDate, o.Total }

```

is translated into

```

customers.Join(orders, c => c.CustomerID, o => o.CustomerID,
    (c, o) => new { c.Name, o.OrderDate, o.Total })

```

The example

```

from c in customers
join o in orders on c.CustomerID equals o.CustomerID into co
let n = co.Count()
where n >= 10
select new { c.Name, OrderCount = n }

```

is translated into

```

from * in customers.
    GroupJoin(orders, c => c.CustomerID, o => o.CustomerID,
        (c, co) => new { c, co })
let n = co.Count()
where n >= 10
select new { c.Name, OrderCount = n }

```

the final translation of which is

```
customers.
GroupJoin(orders, c => c.CustomerID, o => o.CustomerID,
    (c, co) => new { c, co }).
Select(x => new { x, n = x.co.Count() }).
Where(y => y.n >= 10).
Select(y => new { y.x.c.Name, OrderCount = y.n })
```

where `x` and `y` are compiler generated identifiers that are otherwise invisible and inaccessible.

The example

```
from o in orders
orderby o.Customer.Name, o.Total descending
select o
```

has the final translation

```
orders.
OrderBy(o => o.Customer.Name).
ThenByDescending(o => o.Total)
```

Select clauses

A query expression of the form

```
from x in e select v
```

is translated into

```
( e ) . Select ( x => v )
```

except when `v` is the identifier `x`, the translation is simply

```
( e )
```

For example

```
from c in customers.Where(c => c.City == "London")
select c
```

is simply translated into

```
customers.Where(c => c.City == "London")
```

Groupby clauses

A query expression of the form

```
from x in e group v by k
```

is translated into

```
( e ) . GroupBy ( x => k , x => v )
```

except when v is the identifier x , the translation is

```
( e ) . GroupBy ( x => k )
```

The example

```
from c in customers
group c.Name by c.Country
```

is translated into

```
customers.
  GroupBy(c => c.Country, c => c.Name)
```

Transparent identifiers

Certain translations inject range variables with *transparent identifiers* denoted by `*`. Transparent identifiers are not a proper language feature; they exist only as an intermediate step in the query expression translation process.

When a query translation injects a transparent identifier, further translation steps propagate the transparent identifier into anonymous functions and anonymous object initializers. In those contexts, transparent identifiers have the following behavior:

- When a transparent identifier occurs as a parameter in an anonymous function, the members of the associated anonymous type are automatically in scope in the body of the anonymous function.
- When a member with a transparent identifier is in scope, the members of that member are in scope as well.
- When a transparent identifier occurs as a member declarator in an anonymous object initializer, it introduces a member with a transparent identifier.
- In the translation steps described above, transparent identifiers are always introduced together with anonymous types, with the intent of capturing multiple range variables as members of a single object. An implementation of C# is permitted to use a different mechanism than anonymous types to group together multiple range variables. The following translation examples assume that anonymous types are used, and show how transparent identifiers can be translated away.

The example

```
from c in customers
from o in c.Orders
orderby o.Total descending
select new { c.Name, o.Total }
```

is translated into

```
from * in customers.
  SelectMany(c => c.Orders, (c,o) => new { c, o })
orderby o.Total descending
select new { c.Name, o.Total }
```

which is further translated into

```
customers.
SelectMany(c => c.Orders, (c,o) => new { c, o }).
OrderByDescending(* => o.Total).
Select(* => new { c.Name, o.Total })
```

which, when transparent identifiers are erased, is equivalent to

```
customers.
SelectMany(c => c.Orders, (c,o) => new { c, o }).
OrderByDescending(x => x.o.Total).
Select(x => new { x.c.Name, x.o.Total })
```

where `x` is a compiler generated identifier that is otherwise invisible and inaccessible.

The example

```
from c in customers
join o in orders on c.CustomerID equals o.CustomerID
join d in details on o.OrderID equals d.OrderID
join p in products on d.ProductID equals p.ProductID
select new { c.Name, o.OrderDate, p.ProductName }
```

is translated into

```
from * in customers.
    Join(orders, c => c.CustomerID, o => o.CustomerID,
        (c, o) => new { c, o })
join d in details on o.OrderID equals d.OrderID
join p in products on d.ProductID equals p.ProductID
select new { c.Name, o.OrderDate, p.ProductName }
```

which is further reduced to

```
customers.
Join(orders, c => c.CustomerID, o => o.CustomerID, (c, o) => new { c, o }).
Join(details, * => o.OrderID, d => d.OrderID, (*, d) => new { *, d }).
Join(products, * => d.ProductID, p => p.ProductID, (*, p) => new { *, p }).
Select(* => new { c.Name, o.OrderDate, p.ProductName })
```

the final translation of which is

```
customers.
Join(orders, c => c.CustomerID, o => o.CustomerID,
    (c, o) => new { c, o }).
Join(details, x => x.o.OrderID, d => d.OrderID,
    (x, d) => new { x, d }).
Join(products, y => y.d.ProductID, p => p.ProductID,
    (y, p) => new { y, p }).
Select(z => new { z.y.x.c.Name, z.y.x.o.OrderDate, z.p.ProductName })
```

where `x`, `y`, and `z` are compiler generated identifiers that are otherwise invisible and inaccessible.

The query expression pattern

The *Query expression pattern* establishes a pattern of methods that types can implement to support query expressions. Because query expressions are translated to method invocations by means of a syntactic mapping, types have considerable flexibility in how they implement the query expression pattern. For example, the

methods of the pattern can be implemented as instance methods or as extension methods because the two have the same invocation syntax, and the methods can request delegates or expression trees because anonymous functions are convertible to both.

The recommended shape of a generic type `C<T>` that supports the query expression pattern is shown below. A generic type is used in order to illustrate the proper relationships between parameter and result types, but it is possible to implement the pattern for non-generic types as well.

```
delegate R Func<T1,R>(T1 arg1);

delegate R Func<T1,T2,R>(T1 arg1, T2 arg2);

class C
{
    public C<T> Cast<T>();
}

class C<T> : C
{
    public C<T> Where(Func<T,bool> predicate);

    public C<U> Select<U>(Func<T,U> selector);

    public C<V> SelectMany<U,V>(Func<T,C<U>> selector,
        Func<T,U,V> resultSelector);

    public C<V> Join<U,K,V>(C<U> inner, Func<T,K> outerKeySelector,
        Func<U,K> innerKeySelector, Func<T,U,V> resultSelector);

    public C<V> GroupJoin<U,K,V>(C<U> inner, Func<T,K> outerKeySelector,
        Func<U,K> innerKeySelector, Func<T,C<U>,V> resultSelector);

    public O<T> OrderBy<K>(Func<T,K> keySelector);

    public O<T> OrderByDescending<K>(Func<T,K> keySelector);

    public C<G<K,T>> GroupBy<K>(Func<T,K> keySelector);

    public C<G<K,E>> GroupBy<K,E>(Func<T,K> keySelector,
        Func<T,E> elementSelector);
}

class O<T> : C<T>
{
    public O<T> ThenBy<K>(Func<T,K> keySelector);

    public O<T> ThenByDescending<K>(Func<T,K> keySelector);
}

class G<K,T> : C<T>
{
    public K Key { get; }
}
```

The methods above use the generic delegate types `Func<T1,R>` and `Func<T1,T2,R>`, but they could equally well have used other delegate or expression tree types with the same relationships in parameter and result types.

Notice the recommended relationship between `C<T>` and `O<T>` which ensures that the `ThenBy` and `ThenByDescending` methods are available only on the result of an `OrderBy` or `OrderByDescending`. Also notice the recommended shape of the result of `GroupBy` -- a sequence of sequences, where each inner sequence has an additional `Key` property.

The `System.Linq` namespace provides an implementation of the query operator pattern for any type that

implements the `System.Collections.Generic.IEnumerable<T>` interface.

Assignment operators

The assignment operators assign a new value to a variable, a property, an event, or an indexer element.

```
assignment
: unary_expression assignment_operator expression
;

assignment_operator
: '='
| '+='
| '-='
| '*='
| '/='
| '%='
| '&='
| '|='
| '^='
| '<<='
| right_shift_assignment
;
```

The left operand of an assignment must be an expression classified as a variable, a property access, an indexer access, or an event access.

The `=` operator is called the *simple assignment operator*. It assigns the value of the right operand to the variable, property, or indexer element given by the left operand. The left operand of the simple assignment operator may not be an event access (except as described in [Field-like events](#)). The simple assignment operator is described in [Simple assignment](#).

The assignment operators other than the `=` operator are called the *compound assignment operators*. These operators perform the indicated operation on the two operands, and then assign the resulting value to the variable, property, or indexer element given by the left operand. The compound assignment operators are described in [Compound assignment](#).

The `+=` and `-=` operators with an event access expression as the left operand are called the *event assignment operators*. No other assignment operator is valid with an event access as the left operand. The event assignment operators are described in [Event assignment](#).

The assignment operators are right-associative, meaning that operations are grouped from right to left. For example, an expression of the form `a = b = c` is evaluated as `a = (b = c)`.

Simple assignment

The `=` operator is called the simple assignment operator.

If the left operand of a simple assignment is of the form `E.P` or `E[Ei]` where `E` has the compile-time type `dynamic`, then the assignment is dynamically bound ([Dynamic binding](#)). In this case the compile-time type of the assignment expression is `dynamic`, and the resolution described below will take place at run-time based on the run-time type of `E`.

In a simple assignment, the right operand must be an expression that is implicitly convertible to the type of the left operand. The operation assigns the value of the right operand to the variable, property, or indexer element given by the left operand.

The result of a simple assignment expression is the value assigned to the left operand. The result has the same type as the left operand and is always classified as a value.

If the left operand is a property or indexer access, the property or indexer must have a `set` accessor. If this is not the case, a binding-time error occurs.

The run-time processing of a simple assignment of the form `x = y` consists of the following steps:

- If `x` is classified as a variable:
 - `x` is evaluated to produce the variable.
 - `y` is evaluated and, if required, converted to the type of `x` through an implicit conversion ([Implicit conversions](#)).
 - If the variable given by `x` is an array element of a *reference_type*, a run-time check is performed to ensure that the value computed for `y` is compatible with the array instance of which `x` is an element. The check succeeds if `y` is `null`, or if an implicit reference conversion ([Implicit reference conversions](#)) exists from the actual type of the instance referenced by `y` to the actual element type of the array instance containing `x`. Otherwise, a `System.ArrayTypeMismatchException` is thrown.
 - The value resulting from the evaluation and conversion of `y` is stored into the location given by the evaluation of `x`.
- If `x` is classified as a property or indexer access:
 - The instance expression (if `x` is not `static`) and the argument list (if `x` is an indexer access) associated with `x` are evaluated, and the results are used in the subsequent `set` accessor invocation.
 - `y` is evaluated and, if required, converted to the type of `x` through an implicit conversion ([Implicit conversions](#)).
 - The `set` accessor of `x` is invoked with the value computed for `y` as its `value` argument.

The array co-variance rules ([Array covariance](#)) permit a value of an array type `A[]` to be a reference to an instance of an array type `B[]`, provided an implicit reference conversion exists from `B` to `A`. Because of these rules, assignment to an array element of a *reference_type* requires a run-time check to ensure that the value being assigned is compatible with the array instance. In the example

```
string[] sa = new string[10];
object[] oa = sa;

oa[0] = null;           // Ok
oa[1] = "Hello";        // Ok
oa[2] = new ArrayList(); // ArrayTypeMismatchException
```

the last assignment causes a `System.ArrayTypeMismatchException` to be thrown because an instance of `ArrayList` cannot be stored in an element of a `string[]`.

When a property or indexer declared in a *struct_type* is the target of an assignment, the instance expression associated with the property or indexer access must be classified as a variable. If the instance expression is classified as a value, a binding-time error occurs. Because of [Member access](#), the same rule also applies to fields.

Given the declarations:


```

struct Point
{
    int x, y;

    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }

    public int X {
        get { return x; }
        set { x = value; }
    }

    public int Y {
        get { return y; }
        set { y = value; }
    }
}

struct Rectangle
{
    Point a, b;

    public Rectangle(Point a, Point b) {
        this.a = a;
        this.b = b;
    }

    public Point A {
        get { return a; }
        set { a = value; }
    }

    public Point B {
        get { return b; }
        set { b = value; }
    }
}

```

in the example

```

Point p = new Point();
p.X = 100;
p.Y = 100;
Rectangle r = new Rectangle();
r.A = new Point(10, 10);
r.B = p;

```

the assignments to `p.X`, `p.Y`, `r.A`, and `r.B` are permitted because `p` and `r` are variables. However, in the example

```

Rectangle r = new Rectangle();
r.A.X = 10;
r.A.Y = 10;
r.B.X = 100;
r.B.Y = 100;

```

the assignments are all invalid, since `r.A` and `r.B` are not variables.

Compound assignment

If the left operand of a compound assignment is of the form `E.P` or `E[Ei]` where `E` has the compile-time type

`dynamic`, then the assignment is dynamically bound ([Dynamic binding](#)). In this case the compile-time type of the assignment expression is `dynamic`, and the resolution described below will take place at run-time based on the run-time type of `E`.

An operation of the form `x op= y` is processed by applying binary operator overload resolution ([Binary operator overload resolution](#)) as if the operation was written `x op y`. Then,

- If the return type of the selected operator is implicitly convertible to the type of `x`, the operation is evaluated as `x = x op y`, except that `x` is evaluated only once.
- Otherwise, if the selected operator is a predefined operator, if the return type of the selected operator is explicitly convertible to the type of `x`, and if `y` is implicitly convertible to the type of `x` or the operator is a shift operator, then the operation is evaluated as `x = (T)(x op y)`, where `T` is the type of `x`, except that `x` is evaluated only once.
- Otherwise, the compound assignment is invalid, and a binding-time error occurs.

The term "evaluated only once" means that in the evaluation of `x op y`, the results of any constituent expressions of `x` are temporarily saved and then reused when performing the assignment to `x`. For example, in the assignment `A()[B()] += C()`, where `A` is a method returning `int[]`, and `B` and `C` are methods returning `int`, the methods are invoked only once, in the order `A`, `B`, `C`.

When the left operand of a compound assignment is a property access or indexer access, the property or indexer must have both a `get` accessor and a `set` accessor. If this is not the case, a binding-time error occurs.

The second rule above permits `x op= y` to be evaluated as `x = (T)(x op y)` in certain contexts. The rule exists such that the predefined operators can be used as compound operators when the left operand is of type `sbyte`, `byte`, `short`, `ushort`, or `char`. Even when both arguments are of one of those types, the predefined operators produce a result of type `int`, as described in [Binary numeric promotions](#). Thus, without a cast it would not be possible to assign the result to the left operand.

The intuitive effect of the rule for predefined operators is simply that `x op= y` is permitted if both of `x op y` and `x = y` are permitted. In the example

```
byte b = 0;
char ch = '\0';
int i = 0;

b += 1;           // Ok
b += 1000;        // Error, b = 1000 not permitted
b += i;           // Error, b = i not permitted
b += (byte)i;     // Ok

ch += 1;          // Error, ch = 1 not permitted
ch += (char)1;    // Ok
```

the intuitive reason for each error is that a corresponding simple assignment would also have been an error.

This also means that compound assignment operations support lifted operations. In the example

```
int? i = 0;
i += 1;           // Ok
```

the lifted operator `+(int?,int?)` is used.

Event assignment

If the left operand of a `+=` or `-=` operator is classified as an event access, then the expression is evaluated as follows:

- The instance expression, if any, of the event access is evaluated.
- The right operand of the `+=` or `-=` operator is evaluated, and, if required, converted to the type of the left operand through an implicit conversion ([Implicit conversions](#)).
- An event accessor of the event is invoked, with argument list consisting of the right operand, after evaluation and, if necessary, conversion. If the operator was `+=`, the `add` accessor is invoked; if the operator was `-=`, the `remove` accessor is invoked.

An event assignment expression does not yield a value. Thus, an event assignment expression is valid only in the context of a *statement_expression* ([Expression statements](#)).

Expression

An *expression* is either a *non_assignment_expression* or an *assignment*.

```
expression
: non_assignment_expression
| assignment
;

non_assignment_expression
: conditional_expression
| lambda_expression
| query_expression
;
```

Constant expressions

A *constant_expression* is an expression that can be fully evaluated at compile-time.

```
constant_expression
: expression
;
```

A constant expression must be the `null` literal or a value with one of the following types: `sbyte`, `byte`, `short`, `ushort`, `int`, `uint`, `long`, `ulong`, `char`, `float`, `double`, `decimal`, `bool`, `object`, `string`, or any enumeration type. Only the following constructs are permitted in constant expressions:

- Literals (including the `null` literal).
- References to `const` members of class and struct types.
- References to members of enumeration types.
- References to `const` parameters or local variables
- Parenthesized sub-expressions, which are themselves constant expressions.
- Cast expressions, provided the target type is one of the types listed above.
- `checked` and `unchecked` expressions
- Default value expressions
- Nameof expressions
- The predefined `+`, `-`, `!`, and `~` unary operators.
- The predefined `+`, `-`, `*`, `/`, `%`, `<<`, `>>`, `&`, `|`, `^`, `&&`, `||`, `==`, `!=`, `<`, `>`, `<=`, and `>=` binary operators, provided each operand is of a type listed above.
- The `?:` conditional operator.

The following conversions are permitted in constant expressions:

- Identity conversions
- Numeric conversions
- Enumeration conversions
- Constant expression conversions
- Implicit and explicit reference conversions, provided that the source of the conversions is a constant expression that evaluates to the null value.

Other conversions including boxing, unboxing and implicit reference conversions of non-null values are not permitted in constant expressions. For example:

```
class C {
    const object i = 5;           // error: boxing conversion not permitted
    const object str = "hello"; // error: implicit reference conversion
}
```

the initialization of `i` is an error because a boxing conversion is required. The initialization of `str` is an error because an implicit reference conversion from a non-null value is required.

Whenever an expression fulfills the requirements listed above, the expression is evaluated at compile-time. This is true even if the expression is a sub-expression of a larger expression that contains non-constant constructs.

The compile-time evaluation of constant expressions uses the same rules as run-time evaluation of non-constant expressions, except that where run-time evaluation would have thrown an exception, compile-time evaluation causes a compile-time error to occur.

Unless a constant expression is explicitly placed in an `unchecked` context, overflows that occur in integral-type arithmetic operations and conversions during the compile-time evaluation of the expression always cause compile-time errors ([Constant expressions](#)).

Constant expressions occur in the contexts listed below. In these contexts, a compile-time error occurs if an expression cannot be fully evaluated at compile-time.

- Constant declarations ([Constants](#)).
- Enumeration member declarations ([Enum members](#)).
- Default arguments of formal parameter lists ([Method parameters](#))
- `case` labels of a `switch` statement ([The switch statement](#)).
- `goto case` statements ([The goto statement](#)).
- Dimension lengths in an array creation expression ([Array creation expressions](#)) that includes an initializer.
- Attributes ([Attributes](#)).

An implicit constant expression conversion ([Implicit constant expression conversions](#)) permits a constant expression of type `int` to be converted to `sbyte`, `byte`, `short`, `ushort`, `uint`, or `ulong`, provided the value of the constant expression is within the range of the destination type.

Boolean expressions

A *boolean_expression* is an expression that yields a result of type `bool`; either directly or through application of `operator true` in certain contexts as specified in the following.

```
boolean_expression
: expression
;
```

The controlling conditional expression of an *if_statement* ([The if statement](#)), *while_statement* ([The while](#)

[statement](#)), *do_statement* ([The do statement](#)), or *for_statement* ([The for statement](#)) is a *boolean_expression*. The controlling conditional expression of the `?:` operator ([Conditional operator](#)) follows the same rules as a *boolean_expression*, but for reasons of operator precedence is classified as a *conditional_or_expression*.

A *boolean_expression* `E` is required to be able to produce a value of type `bool`, as follows:

- If `E` is implicitly convertible to `bool` then at runtime that implicit conversion is applied.
- Otherwise, unary operator overload resolution ([Unary operator overload resolution](#)) is used to find a unique best implementation of operator `true` on `E`, and that implementation is applied at runtime.
- If no such operator is found, a binding-time error occurs.

The `DBBbool` struct type in [Database boolean type](#) provides an example of a type that implements `operator true` and `operator false`.

Statements

12/28/2021 • 54 minutes to read • [Edit Online](#)

C# provides a variety of statements. Most of these statements will be familiar to developers who have programmed in C and C++.

```
statement
: labeled_statement
| declaration_statement
| embedded_statement
;

embedded_statement
: block
| empty_statement
| expression_statement
| selection_statement
| iteration_statement
| jump_statement
| try_statement
| checked_statement
| unchecked_statement
| lock_statement
| using_statement
| yield_statement
| embedded_statement_unsafe
;
```

The *embedded_statement* nonterminal is used for statements that appear within other statements. The use of *embedded_statement* rather than *statement* excludes the use of declaration statements and labeled statements in these contexts. The example

```
void F(bool b) {
    if (b)
        int i = 44;
}
```

results in a compile-time error because an `if` statement requires an *embedded_statement* rather than a *statement* for its if branch. If this code were permitted, then the variable `i` would be declared, but it could never be used. Note, however, that by placing `i`'s declaration in a block, the example is valid.

End points and reachability

Every statement has an *end point*. In intuitive terms, the end point of a statement is the location that immediately follows the statement. The execution rules for composite statements (statements that contain embedded statements) specify the action that is taken when control reaches the end point of an embedded statement. For example, when control reaches the end point of a statement in a block, control is transferred to the next statement in the block.

If a statement can possibly be reached by execution, the statement is said to be *reachable*. Conversely, if there is no possibility that a statement will be executed, the statement is said to be *unreachable*.

In the example

```
void F() {
    Console.WriteLine("reachable");
    goto Label;
    Console.WriteLine("unreachable");
Label:
    Console.WriteLine("reachable");
}
```

the second invocation of `Console.WriteLine` is unreachable because there is no possibility that the statement will be executed.

A warning is reported if the compiler determines that a statement is unreachable. It is specifically not an error for a statement to be unreachable.

To determine whether a particular statement or end point is reachable, the compiler performs flow analysis according to the reachability rules defined for each statement. The flow analysis takes into account the values of constant expressions ([Constant expressions](#)) that control the behavior of statements, but the possible values of non-constant expressions are not considered. In other words, for purposes of control flow analysis, a non-constant expression of a given type is considered to have any possible value of that type.

In the example

```
void F() {
    const int i = 1;
    if (i == 2) Console.WriteLine("unreachable");
}
```

the boolean expression of the `if` statement is a constant expression because both operands of the `==` operator are constants. As the constant expression is evaluated at compile-time, producing the value `false`, the `Console.WriteLine` invocation is considered unreachable. However, if `i` is changed to be a local variable

```
void F() {
    int i = 1;
    if (i == 2) Console.WriteLine("reachable");
}
```

the `Console.WriteLine` invocation is considered reachable, even though, in reality, it will never be executed.

The *block* of a function member is always considered reachable. By successively evaluating the reachability rules of each statement in a block, the reachability of any given statement can be determined.

In the example

```
void F(int x) {
    Console.WriteLine("start");
    if (x < 0) Console.WriteLine("negative");
}
```

the reachability of the second `Console.WriteLine` is determined as follows:

- The first `Console.WriteLine` expression statement is reachable because the block of the `F` method is reachable.
- The end point of the first `Console.WriteLine` expression statement is reachable because that statement is reachable.
- The `if` statement is reachable because the end point of the first `Console.WriteLine` expression statement is reachable.

- The second `Console.WriteLine` expression statement is reachable because the boolean expression of the `if` statement does not have the constant value `false`.

There are two situations in which it is a compile-time error for the end point of a statement to be reachable:

- Because the `switch` statement does not permit a switch section to "fall through" to the next switch section, it is a compile-time error for the end point of the statement list of a switch section to be reachable. If this error occurs, it is typically an indication that a `break` statement is missing.
- It is a compile-time error for the end point of the block of a function member that computes a value to be reachable. If this error occurs, it typically is an indication that a `return` statement is missing.

Blocks

A *block* permits multiple statements to be written in contexts where a single statement is allowed.

```
block
    : '{' statement_list? '}'
    ;
```

A *block* consists of an optional *statement_list* ([Statement lists](#)), enclosed in braces. If the statement list is omitted, the block is said to be empty.

A block may contain declaration statements ([Declaration statements](#)). The scope of a local variable or constant declared in a block is the block.

A block is executed as follows:

- If the block is empty, control is transferred to the end point of the block.
- If the block is not empty, control is transferred to the statement list. When and if control reaches the end point of the statement list, control is transferred to the end point of the block.

The statement list of a block is reachable if the block itself is reachable.

The end point of a block is reachable if the block is empty or if the end point of the statement list is reachable.

A *block* that contains one or more `yield` statements ([The yield statement](#)) is called an iterator block. Iterator blocks are used to implement function members as iterators ([Iterators](#)). Some additional restrictions apply to iterator blocks:

- It is a compile-time error for a `return` statement to appear in an iterator block (but `yield return` statements are permitted).
- It is a compile-time error for an iterator block to contain an unsafe context ([Unsafe contexts](#)). An iterator block always defines a safe context, even when its declaration is nested in an unsafe context.

Statement lists

A ***statement list*** consists of one or more statements written in sequence. Statement lists occur in *blocks* ([Blocks](#)) and in *switch_blocks* ([The switch statement](#)).

```
statement_list
    : statement+
    ;
```

A statement list is executed by transferring control to the first statement. When and if control reaches the end point of a statement, control is transferred to the next statement. When and if control reaches the end point of the last statement, control is transferred to the end point of the statement list.

A statement in a statement list is reachable if at least one of the following is true:

- The statement is the first statement and the statement list itself is reachable.
- The end point of the preceding statement is reachable.
- The statement is a labeled statement and the label is referenced by a reachable `goto` statement.

The end point of a statement list is reachable if the end point of the last statement in the list is reachable.

The empty statement

An *empty_statement* does nothing.

```
empty_statement
: ';'
;
```

An empty statement is used when there are no operations to perform in a context where a statement is required.

Execution of an empty statement simply transfers control to the end point of the statement. Thus, the end point of an empty statement is reachable if the empty statement is reachable.

An empty statement can be used when writing a `while` statement with a null body:

```
bool ProcessMessage() {...}

void ProcessMessages() {
    while (ProcessMessage())
        ;
}
```

Also, an empty statement can be used to declare a label just before the closing "`}`" of a block:

```
void F() {
    ...
    if (done) goto exit;
    ...
    exit: ;
}
```

Labeled statements

A *labeled_statement* permits a statement to be prefixed by a label. Labeled statements are permitted in blocks, but are not permitted as embedded statements.

```
labeled_statement
: identifier ':' statement
;
```

A labeled statement declares a label with the name given by the *identifier*. The scope of a label is the whole block in which the label is declared, including any nested blocks. It is a compile-time error for two labels with the same name to have overlapping scopes.

A label can be referenced from `goto` statements ([The goto statement](#)) within the scope of the label. This means that `goto` statements can transfer control within blocks and out of blocks, but never into blocks.

Labels have their own declaration space and do not interfere with other identifiers. The example

```
int F(int x) {
    if (x >= 0) goto x;
    x = -x;
    x: return x;
}
```

is valid and uses the name `x` as both a parameter and a label.

Execution of a labeled statement corresponds exactly to execution of the statement following the label.

In addition to the reachability provided by normal flow of control, a labeled statement is reachable if the label is referenced by a reachable `goto` statement. (Exception: If a `goto` statement is inside a `try` that includes a `finally` block, and the labeled statement is outside the `try`, and the end point of the `finally` block is unreachable, then the labeled statement is not reachable from that `goto` statement.)

Declaration statements

A *declaration_statement* declares a local variable or constant. Declaration statements are permitted in blocks, but are not permitted as embedded statements.

```
declaration_statement
: local_variable_declaration ';'
| local_constant_declaration ';'
;
```

Local variable declarations

A *local_variable_declaration* declares one or more local variables.

```
local_variable_declaration
: local_variable_type local_variable_declarators
;

local_variable_type
: type
| 'var'
;

local_variable_declarators
: local_variable_declarator
| local_variable_declarators ',' local_variable_declarator
;

local_variable_declarator
: identifier
| identifier '=' local_variable_initializer
;

local_variable_initializer
: expression
| array_initializer
| local_variable_initializer_unsafe
;
```

The *local_variable_type* of a *local_variable_declaration* either directly specifies the type of the variables introduced by the declaration, or indicates with the identifier `var` that the type should be inferred based on an initializer. The type is followed by a list of *local_variable_declarators*, each of which introduces a new variable. A *local_variable_declarator* consists of an *identifier* that names the variable, optionally followed by an `=` token and a *local_variable_initializer* that gives the initial value of the variable.

In the context of a local variable declaration, the identifier `var` acts as a contextual keyword ([Keywords](#)). When the *local_variable_type* is specified as `var` and no type named `var` is in scope, the declaration is an **implicitly typed local variable declaration**, whose type is inferred from the type of the associated initializer expression. Implicitly typed local variable declarations are subject to the following restrictions:

- The *local_variable_declaration* cannot include multiple *local_variable_declarators*.
- The *local_variable_declarator* must include a *local_variable_initializer*.
- The *local_variable_initializer* must be an *expression*.
- The initializer *expression* must have a compile-time type.
- The initializer *expression* cannot refer to the declared variable itself

The following are examples of incorrect implicitly typed local variable declarations:

```
var x;           // Error, no initializer to infer type from
var y = {1, 2, 3}; // Error, array initializer not permitted
var z = null;    // Error, null does not have a type
var u = x => x + 1; // Error, anonymous functions do not have a type
var v = v++;     // Error, initializer cannot refer to variable itself
```

The value of a local variable is obtained in an expression using a *simple_name* ([Simple names](#)), and the value of a local variable is modified using an *assignment* ([Assignment operators](#)). A local variable must be definitely assigned ([Definite assignment](#)) at each location where its value is obtained.

The scope of a local variable declared in a *local_variable_declaration* is the block in which the declaration occurs. It is an error to refer to a local variable in a textual position that precedes the *local_variable_declarator* of the local variable. Within the scope of a local variable, it is a compile-time error to declare another local variable or constant with the same name.

A local variable declaration that declares multiple variables is equivalent to multiple declarations of single variables with the same type. Furthermore, a variable initializer in a local variable declaration corresponds exactly to an assignment statement that is inserted immediately after the declaration.

The example

```
void F() {
    int x = 1, y, z = x * 2;
}
```

corresponds exactly to

```
void F() {
    int x; x = 1;
    int y;
    int z; z = x * 2;
}
```

In an implicitly typed local variable declaration, the type of the local variable being declared is taken to be the same as the type of the expression used to initialize the variable. For example:

```
var i = 5;
var s = "Hello";
var d = 1.0;
var numbers = new int[] {1, 2, 3};
var orders = new Dictionary<int, Order>();
```

The implicitly typed local variable declarations above are precisely equivalent to the following explicitly typed declarations:

```
int i = 5;
string s = "Hello";
double d = 1.0;
int[] numbers = new int[] {1, 2, 3};
Dictionary<int,Order> orders = new Dictionary<int,Order>();
```

Local constant declarations

A *local_constant_declaration* declares one or more local constants.

```
local_constant_declaration
    : 'const' type constant_declarators
    ;

constant_declarators
    : constant_declarator (',' constant_declarator)*
    ;

constant_declarator
    : identifier '=' constant_expression
    ;
```

The *type* of a *local_constant_declaration* specifies the type of the constants introduced by the declaration. The type is followed by a list of *constant_declarators*, each of which introduces a new constant. A *constant_declarator* consists of an *identifier* that names the constant, followed by an "=" token, followed by a *constant_expression* ([Constant expressions](#)) that gives the value of the constant.

The *type* and *constant_expression* of a local constant declaration must follow the same rules as those of a constant member declaration ([Constants](#)).

The value of a local constant is obtained in an expression using a *simple_name* ([Simple names](#)).

The scope of a local constant is the block in which the declaration occurs. It is an error to refer to a local constant in a textual position that precedes its *constant_declarator*. Within the scope of a local constant, it is a compile-time error to declare another local variable or constant with the same name.

A local constant declaration that declares multiple constants is equivalent to multiple declarations of single constants with the same type.

Expression statements

An *expression_statement* evaluates a given expression. The value computed by the expression, if any, is discarded.

```

expression_statement
: statement_expression ';'
;

statement_expression
: invocation_expression
| null_conditional_invocation_expression
| object_creation_expression
| assignment
| post_increment_expression
| post_decrement_expression
| pre_increment_expression
| pre_decrement_expression
| await_expression
;

```

Not all expressions are permitted as statements. In particular, expressions such as `x + y` and `x == 1` that merely compute a value (which will be discarded), are not permitted as statements.

Execution of an *expression_statement* evaluates the contained expression and then transfers control to the end point of the *expression_statement*. The end point of an *expression_statement* is reachable if that *expression_statement* is reachable.

Selection statements

Selection statements select one of a number of possible statements for execution based on the value of some expression.

```

selection_statement
: if_statement
| switch_statement
;

```

The if statement

The `if` statement selects a statement for execution based on the value of a boolean expression.

```

if_statement
: 'if' '(' boolean_expression ')' embedded_statement
| 'if' '(' boolean_expression ')' embedded_statement 'else' embedded_statement
;

```

An `else` part is associated with the lexically nearest preceding `if` that is allowed by the syntax. Thus, an `if` statement of the form

```
if (x) if (y) F(); else G();
```

is equivalent to

```

if (x) {
    if (y) {
        F();
    }
    else {
        G();
    }
}

```

An `if` statement is executed as follows:

- The *boolean_expression* ([Boolean expressions](#)) is evaluated.
- If the boolean expression yields `true`, control is transferred to the first embedded statement. When and if control reaches the end point of that statement, control is transferred to the end point of the `if` statement.
- If the boolean expression yields `false` and if an `else` part is present, control is transferred to the second embedded statement. When and if control reaches the end point of that statement, control is transferred to the end point of the `if` statement.
- If the boolean expression yields `false` and if an `else` part is not present, control is transferred to the end point of the `if` statement.

The first embedded statement of an `if` statement is reachable if the `if` statement is reachable and the boolean expression does not have the constant value `false`.

The second embedded statement of an `if` statement, if present, is reachable if the `if` statement is reachable and the boolean expression does not have the constant value `true`.

The end point of an `if` statement is reachable if the end point of at least one of its embedded statements is reachable. In addition, the end point of an `if` statement with no `else` part is reachable if the `if` statement is reachable and the boolean expression does not have the constant value `true`.

The switch statement

The switch statement selects for execution a statement list having an associated switch label that corresponds to the value of the switch expression.

```
switch_statement
: 'switch' '(' expression ')' switch_block
;

switch_block
: '{' switch_section* '}'
;

switch_section
: switch_label+ statement_list
;

switch_label
: 'case' constant_expression ':'
| 'default' ':'
;
```

A *switch_statement* consists of the keyword `switch`, followed by a parenthesized expression (called the switch expression), followed by a *switch_block*. The *switch_block* consists of zero or more *switch_sections*, enclosed in braces. Each *switch_section* consists of one or more *switch_labels* followed by a *statement_list* ([Statement lists](#)).

The *governing type* of a `switch` statement is established by the switch expression.

- If the type of the switch expression is `sbyte`, `byte`, `short`, `ushort`, `int`, `uint`, `long`, `ulong`, `bool`, `char`, `string`, or an *enum_type*, or if it is the nullable type corresponding to one of these types, then that is the governing type of the `switch` statement.
- Otherwise, exactly one user-defined implicit conversion ([User-defined conversions](#)) must exist from the type of the switch expression to one of the following possible governing types: `sbyte`, `byte`, `short`, `ushort`, `int`, `uint`, `long`, `ulong`, `char`, `string`, or, a nullable type corresponding to one of those types.
- Otherwise, if no such implicit conversion exists, or if more than one such implicit conversion exists, a compile-time error occurs.

The constant expression of each `case` label must denote a value that is implicitly convertible ([Implicit conversions](#)) to the governing type of the `switch` statement. A compile-time error occurs if two or more `case` labels in the same `switch` statement specify the same constant value.

There can be at most one `default` label in a switch statement.

A `switch` statement is executed as follows:

- The switch expression is evaluated and converted to the governing type.
- If one of the constants specified in a `case` label in the same `switch` statement is equal to the value of the switch expression, control is transferred to the statement list following the matched `case` label.
- If none of the constants specified in `case` labels in the same `switch` statement is equal to the value of the switch expression, and if a `default` label is present, control is transferred to the statement list following the `default` label.
- If none of the constants specified in `case` labels in the same `switch` statement is equal to the value of the switch expression, and if no `default` label is present, control is transferred to the end point of the `switch` statement.

If the end point of the statement list of a switch section is reachable, a compile-time error occurs. This is known as the "no fall through" rule. The example

```
switch (i) {  
  case 0:  
    CaseZero();  
    break;  
  case 1:  
    CaseOne();  
    break;  
  default:  
    CaseOthers();  
    break;  
}
```

is valid because no switch section has a reachable end point. Unlike C and C++, execution of a switch section is not permitted to "fall through" to the next switch section, and the example

```
switch (i) {  
  case 0:  
    CaseZero();  
  case 1:  
    CaseZeroOrOne();  
  default:  
    CaseAny();  
}
```

results in a compile-time error. When execution of a switch section is to be followed by execution of another switch section, an explicit `goto case` or `goto default` statement must be used:

```

switch (i) {
case 0:
    CaseZero();
    goto case 1;
case 1:
    CaseZeroOrOne();
    goto default;
default:
    CaseAny();
    break;
}

```

Multiple labels are permitted in a *switch_section*. The example

```

switch (i) {
case 0:
    CaseZero();
    break;
case 1:
    CaseOne();
    break;
case 2:
default:
    CaseTwo();
    break;
}

```

is valid. The example does not violate the "no fall through" rule because the labels `case 2:` and `default:` are part of the same *switch_section*.

The "no fall through" rule prevents a common class of bugs that occur in C and C++ when `break` statements are accidentally omitted. In addition, because of this rule, the switch sections of a `switch` statement can be arbitrarily rearranged without affecting the behavior of the statement. For example, the sections of the `switch` statement above can be reversed without affecting the behavior of the statement:

```

switch (i) {
default:
    CaseAny();
    break;
case 1:
    CaseZeroOrOne();
    goto default;
case 0:
    CaseZero();
    goto case 1;
}

```

The statement list of a switch section typically ends in a `break`, `goto case`, or `goto default` statement, but any construct that renders the end point of the statement list unreachable is permitted. For example, a `while` statement controlled by the boolean expression `true` is known to never reach its end point. Likewise, a `throw` or `return` statement always transfers control elsewhere and never reaches its end point. Thus, the following example is valid:


```

switch (i) {
case 0:
    while (true) F();
case 1:
    throw new ArgumentException();
case 2:
    return;
}

```

The governing type of a `switch` statement may be the type `string`. For example:

```

void DoCommand(string command) {
    switch (command.ToLower()) {
case "run":
    DoRun();
    break;
case "save":
    DoSave();
    break;
case "quit":
    DoQuit();
    break;
default:
    InvalidCommand(command);
    break;
    }
}

```

Like the string equality operators ([String equality operators](#)), the `switch` statement is case sensitive and will execute a given switch section only if the switch expression string exactly matches a `case` label constant.

When the governing type of a `switch` statement is `string`, the value `null` is permitted as a case label constant.

The *statement_lists* of a *switch_block* may contain declaration statements ([Declaration statements](#)). The scope of a local variable or constant declared in a switch block is the switch block.

The statement list of a given switch section is reachable if the `switch` statement is reachable and at least one of the following is true:

- The switch expression is a non-constant value.
- The switch expression is a constant value that matches a `case` label in the switch section.
- The switch expression is a constant value that doesn't match any `case` label, and the switch section contains the `default` label.
- A switch label of the switch section is referenced by a reachable `goto case` or `goto default` statement.

The end point of a `switch` statement is reachable if at least one of the following is true:

- The `switch` statement contains a reachable `break` statement that exits the `switch` statement.
- The `switch` statement is reachable, the switch expression is a non-constant value, and no `default` label is present.
- The `switch` statement is reachable, the switch expression is a constant value that doesn't match any `case` label, and no `default` label is present.

Iteration statements

Iteration statements repeatedly execute an embedded statement.

```
iteration_statement
: while_statement
| do_statement
| for_statement
| foreach_statement
;
```

The while statement

The `while` statement conditionally executes an embedded statement zero or more times.

```
while_statement
: 'while' '(' boolean_expression ')' embedded_statement
;
```

A `while` statement is executed as follows:

- The *boolean_expression* ([Boolean expressions](#)) is evaluated.
- If the boolean expression yields `true`, control is transferred to the embedded statement. When and if control reaches the end point of the embedded statement (possibly from execution of a `continue` statement), control is transferred to the beginning of the `while` statement.
- If the boolean expression yields `false`, control is transferred to the end point of the `while` statement.

Within the embedded statement of a `while` statement, a `break` statement ([The break statement](#)) may be used to transfer control to the end point of the `while` statement (thus ending iteration of the embedded statement), and a `continue` statement ([The continue statement](#)) may be used to transfer control to the end point of the embedded statement (thus performing another iteration of the `while` statement).

The embedded statement of a `while` statement is reachable if the `while` statement is reachable and the boolean expression does not have the constant value `false`.

The end point of a `while` statement is reachable if at least one of the following is true:

- The `while` statement contains a reachable `break` statement that exits the `while` statement.
- The `while` statement is reachable and the boolean expression does not have the constant value `true`.

The do statement

The `do` statement conditionally executes an embedded statement one or more times.

```
do_statement
: 'do' embedded_statement 'while' '(' boolean_expression ')' ';'
;
```

A `do` statement is executed as follows:

- Control is transferred to the embedded statement.
- When and if control reaches the end point of the embedded statement (possibly from execution of a `continue` statement), the *boolean_expression* ([Boolean expressions](#)) is evaluated. If the boolean expression yields `true`, control is transferred to the beginning of the `do` statement. Otherwise, control is transferred to the end point of the `do` statement.

Within the embedded statement of a `do` statement, a `break` statement ([The break statement](#)) may be used to transfer control to the end point of the `do` statement (thus ending iteration of the embedded statement), and a `continue` statement ([The continue statement](#)) may be used to transfer control to the end point of the embedded statement.

The embedded statement of a `do` statement is reachable if the `do` statement is reachable.

The end point of a `do` statement is reachable if at least one of the following is true:

- The `do` statement contains a reachable `break` statement that exits the `do` statement.
- The end point of the embedded statement is reachable and the boolean expression does not have the constant value `true`.

The for statement

The `for` statement evaluates a sequence of initialization expressions and then, while a condition is true, repeatedly executes an embedded statement and evaluates a sequence of iteration expressions.

```
for_statement
: 'for' '(' for_initializer? ';' for_condition? ';' for_iterator? ')' embedded_statement
;

for_initializer
: local_variable_declaration
| statement_expression_list
;

for_condition
: boolean_expression
;

for_iterator
: statement_expression_list
;

statement_expression_list
: statement_expression (',' statement_expression)*
;
```

The *for_initializer*, if present, consists of either a *local_variable_declaration* ([Local variable declarations](#)) or a list of *statement_expressions* ([Expression statements](#)) separated by commas. The scope of a local variable declared by a *for_initializer* starts at the *local_variable_declarator* for the variable and extends to the end of the embedded statement. The scope includes the *for_condition* and the *for_iterator*.

The *for_condition*, if present, must be a *boolean_expression* ([Boolean expressions](#)).

The *for_iterator*, if present, consists of a list of *statement_expressions* ([Expression statements](#)) separated by commas.

A for statement is executed as follows:

- If a *for_initializer* is present, the variable initializers or statement expressions are executed in the order they are written. This step is only performed once.
- If a *for_condition* is present, it is evaluated.
- If the *for_condition* is not present or if the evaluation yields `true`, control is transferred to the embedded statement. When and if control reaches the end point of the embedded statement (possibly from execution of a `continue` statement), the expressions of the *for_iterator*, if any, are evaluated in sequence, and then another iteration is performed, starting with evaluation of the *for_condition* in the step above.
- If the *for_condition* is present and the evaluation yields `false`, control is transferred to the end point of the `for` statement.

Within the embedded statement of a `for` statement, a `break` statement ([The break statement](#)) may be used to transfer control to the end point of the `for` statement (thus ending iteration of the embedded statement), and a `continue` statement ([The continue statement](#)) may be used to transfer control to the end point of the embedded

statement (thus executing the *for_iterator* and performing another iteration of the `for` statement, starting with the *for_condition*).

The embedded statement of a `for` statement is reachable if one of the following is true:

- The `for` statement is reachable and no *for_condition* is present.
- The `for` statement is reachable and a *for_condition* is present and does not have the constant value `false`.

The end point of a `for` statement is reachable if at least one of the following is true:

- The `for` statement contains a reachable `break` statement that exits the `for` statement.
- The `for` statement is reachable and a *for_condition* is present and does not have the constant value `true`.

The foreach statement

The `foreach` statement enumerates the elements of a collection, executing an embedded statement for each element of the collection.

```
foreach_statement
: 'foreach' '(' local_variable_type identifier 'in' expression ')' embedded_statement
;
```

The *type* and *identifier* of a `foreach` statement declare the **iteration variable** of the statement. If the `var` identifier is given as the *local_variable_type*, and no type named `var` is in scope, the iteration variable is said to be an **implicitly typed iteration variable**, and its type is taken to be the element type of the `foreach` statement, as specified below. The iteration variable corresponds to a read-only local variable with a scope that extends over the embedded statement. During execution of a `foreach` statement, the iteration variable represents the collection element for which an iteration is currently being performed. A compile-time error occurs if the embedded statement attempts to modify the iteration variable (via assignment or the `++` and `--` operators) or pass the iteration variable as a `ref` or `out` parameter.

In the following, for brevity, `IEnumerable`, `IEnumerator`, `IEnumerable<T>` and `IEnumerator<T>` refer to the corresponding types in the namespaces `System.Collections` and `System.Collections.Generic`.

The compile-time processing of a `foreach` statement first determines the **collection type**, **enumerator type** and **element type** of the expression. This determination proceeds as follows:

- If the type `x` of *expression* is an array type then there is an implicit reference conversion from `x` to the `IEnumerable` interface (since `System.Array` implements this interface). The **collection type** is the `IEnumerable` interface, the **enumerator type** is the `IEnumerator` interface and the **element type** is the element type of the array type `x`.
- If the type `x` of *expression* is `dynamic` then there is an implicit conversion from *expression* to the `IEnumerable` interface ([Implicit dynamic conversions](#)). The **collection type** is the `IEnumerable` interface and the **enumerator type** is the `IEnumerator` interface. If the `var` identifier is given as the *local_variable_type* then the **element type** is `dynamic`, otherwise it is `object`.
- Otherwise, determine whether the type `x` has an appropriate `GetEnumerator` method:
 - Perform member lookup on the type `x` with identifier `GetEnumerator` and no type arguments. If the member lookup does not produce a match, or it produces an ambiguity, or produces a match that is not a method group, check for an enumerable interface as described below. It is recommended that a warning be issued if member lookup produces anything except a method group or no match.
 - Perform overload resolution using the resulting method group and an empty argument list. If overload resolution results in no applicable methods, results in an ambiguity, or results in a single best method but that method is either static or not public, check for an enumerable interface as described

below. It is recommended that a warning be issued if overload resolution produces anything except an unambiguous public instance method or no applicable methods.

- If the return type `E` of the `GetEnumerator` method is not a class, struct or interface type, an error is produced and no further steps are taken.
- Member lookup is performed on `E` with the identifier `Current` and no type arguments. If the member lookup produces no match, the result is an error, or the result is anything except a public instance property that permits reading, an error is produced and no further steps are taken.
- Member lookup is performed on `E` with the identifier `MoveNext` and no type arguments. If the member lookup produces no match, the result is an error, or the result is anything except a method group, an error is produced and no further steps are taken.
- Overload resolution is performed on the method group with an empty argument list. If overload resolution results in no applicable methods, results in an ambiguity, or results in a single best method but that method is either static or not public, or its return type is not `bool`, an error is produced and no further steps are taken.
- The *collection type* is `X`, the *enumerator type* is `E`, and the *element type* is the type of the `Current` property.
- Otherwise, check for an enumerable interface:
 - If among all the types `Ti` for which there is an implicit conversion from `X` to `IEnumerable<Ti>`, there is a unique type `T` such that `T` is not `dynamic` and for all the other `Ti` there is an implicit conversion from `IEnumerable<T>` to `IEnumerable<Ti>`, then the *collection type* is the interface `IEnumerable<T>`, the *enumerator type* is the interface `IEnumerator<T>`, and the *element type* is `T`.
 - Otherwise, if there is more than one such type `T`, then an error is produced and no further steps are taken.
 - Otherwise, if there is an implicit conversion from `X` to the `System.Collections.IEnumerable` interface, then the *collection type* is this interface, the *enumerator type* is the interface `System.Collections.IEnumerator`, and the *element type* is `object`.
 - Otherwise, an error is produced and no further steps are taken.

The above steps, if successful, unambiguously produce a collection type `C`, enumerator type `E` and element type `T`. A `foreach` statement of the form

```
foreach (V v in x) embedded_statement
```

is then expanded to:

```
{
    E e = ((C)(x)).GetEnumerator();
    try {
        while (e.MoveNext()) {
            V v = (V)(T)e.Current;
            embedded_statement
        }
    }
    finally {
        ... // Dispose e
    }
}
```

The variable `e` is not visible to or accessible to the expression `x` or the embedded statement or any other source code of the program. The variable `v` is read-only in the embedded statement. If there is not an explicit conversion ([Explicit conversions](#)) from `T` (the element type) to `V` (the *local_variable_type* in the `foreach` statement), an error is produced and no further steps are taken. If `x` has the value `null`, a

`System.NullReferenceException` is thrown at run-time.

An implementation is permitted to implement a given foreach-statement differently, e.g. for performance reasons, as long as the behavior is consistent with the above expansion.

The placement of `v` inside the while loop is important for how it is captured by any anonymous function occurring in the *embedded_statement*.

For example:

```
int[] values = { 7, 9, 13 };
Action f = null;

foreach (var value in values)
{
    if (f == null) f = () => Console.WriteLine("First value: " + value);
}

f();
```

If `v` was declared outside of the while loop, it would be shared among all iterations, and its value after the for loop would be the final value, `13`, which is what the invocation of `f` would print. Instead, because each iteration has its own variable `v`, the one captured by `f` in the first iteration will continue to hold the value `7`, which is what will be printed. (Note: earlier versions of C# declared `v` outside of the while loop.)

The body of the finally block is constructed according to the following steps:

- If there is an implicit conversion from `E` to the `System.IDisposable` interface, then
 - If `E` is a non-nullable value type then the finally clause is expanded to the semantic equivalent of:

```
finally {
    ((System.IDisposable)e).Dispose();
}
```

- Otherwise the finally clause is expanded to the semantic equivalent of:

```
finally {
    if (e != null) ((System.IDisposable)e).Dispose();
}
```

except that if `E` is a value type, or a type parameter instantiated to a value type, then the cast of `e` to `System.IDisposable` will not cause boxing to occur.

- Otherwise, if `E` is a sealed type, the finally clause is expanded to an empty block:

```
finally {
}
```

- Otherwise, the finally clause is expanded to:

```
finally {
    System.IDisposable d = e as System.IDisposable;
    if (d != null) d.Dispose();
}
```

The local variable `d` is not visible to or accessible to any user code. In particular, it does not conflict with any other variable whose scope includes the finally block.

The order in which `foreach` traverses the elements of an array, is as follows: For single-dimensional arrays elements are traversed in increasing index order, starting with index `0` and ending with index `Length - 1`. For multi-dimensional arrays, elements are traversed such that the indices of the rightmost dimension are increased first, then the next left dimension, and so on to the left.

The following example prints out each value in a two-dimensional array, in element order:

```
using System;

class Test
{
    static void Main() {
        double[,] values = {
            {1.2, 2.3, 3.4, 4.5},
            {5.6, 6.7, 7.8, 8.9}
        };

        foreach (double elementValue in values)
            Console.Write("{0} ", elementValue);

        Console.WriteLine();
    }
}
```

The output produced is as follows:

```
1.2 2.3 3.4 4.5 5.6 6.7 7.8 8.9
```

In the example

```
int[] numbers = { 1, 3, 5, 7, 9 };
foreach (var n in numbers) Console.WriteLine(n);
```

the type of `n` is inferred to be `int`, the element type of `numbers`.

Jump statements

Jump statements unconditionally transfer control.

```
jump_statement
: break_statement
| continue_statement
| goto_statement
| return_statement
| throw_statement
;
```

The location to which a jump statement transfers control is called the *target* of the jump statement.

When a jump statement occurs within a block, and the target of that jump statement is outside that block, the jump statement is said to *exit* the block. While a jump statement may transfer control out of a block, it can never transfer control into a block.

Execution of jump statements is complicated by the presence of intervening `try` statements. In the absence of

such `try` statements, a jump statement unconditionally transfers control from the jump statement to its target. In the presence of such intervening `try` statements, execution is more complex. If the jump statement exits one or more `try` blocks with associated `finally` blocks, control is initially transferred to the `finally` block of the innermost `try` statement. When and if control reaches the end point of a `finally` block, control is transferred to the `finally` block of the next enclosing `try` statement. This process is repeated until the `finally` blocks of all intervening `try` statements have been executed.

In the example

```
using System;

class Test
{
    static void Main() {
        while (true) {
            try {
                try {
                    Console.WriteLine("Before break");
                    break;
                }
                finally {
                    Console.WriteLine("Innermost finally block");
                }
            }
            finally {
                Console.WriteLine("Outermost finally block");
            }
        }
        Console.WriteLine("After break");
    }
}
```

the `finally` blocks associated with two `try` statements are executed before control is transferred to the target of the jump statement.

The output produced is as follows:

```
Before break
Innermost finally block
Outermost finally block
After break
```

The break statement

The `break` statement exits the nearest enclosing `switch`, `while`, `do`, `for`, or `foreach` statement.

```
break_statement
: 'break' ';'
;
```

The target of a `break` statement is the end point of the nearest enclosing `switch`, `while`, `do`, `for`, or `foreach` statement. If a `break` statement is not enclosed by a `switch`, `while`, `do`, `for`, or `foreach` statement, a compile-time error occurs.

When multiple `switch`, `while`, `do`, `for`, or `foreach` statements are nested within each other, a `break` statement applies only to the innermost statement. To transfer control across multiple nesting levels, a `goto` statement ([The goto statement](#)) must be used.

A `break` statement cannot exit a `finally` block ([The try statement](#)). When a `break` statement occurs within a

`finally` block, the target of the `break` statement must be within the same `finally` block; otherwise, a compile-time error occurs.

A `break` statement is executed as follows:

- If the `break` statement exits one or more `try` blocks with associated `finally` blocks, control is initially transferred to the `finally` block of the innermost `try` statement. When and if control reaches the end point of a `finally` block, control is transferred to the `finally` block of the next enclosing `try` statement. This process is repeated until the `finally` blocks of all intervening `try` statements have been executed.
- Control is transferred to the target of the `break` statement.

Because a `break` statement unconditionally transfers control elsewhere, the end point of a `break` statement is never reachable.

The `continue` statement

The `continue` statement starts a new iteration of the nearest enclosing `while`, `do`, `for`, or `foreach` statement.

```
continue_statement
: 'continue' ';'
;
```

The target of a `continue` statement is the end point of the embedded statement of the nearest enclosing `while`, `do`, `for`, or `foreach` statement. If a `continue` statement is not enclosed by a `while`, `do`, `for`, or `foreach` statement, a compile-time error occurs.

When multiple `while`, `do`, `for`, or `foreach` statements are nested within each other, a `continue` statement applies only to the innermost statement. To transfer control across multiple nesting levels, a `goto` statement ([The `goto` statement](#)) must be used.

A `continue` statement cannot exit a `finally` block ([The `try` statement](#)). When a `continue` statement occurs within a `finally` block, the target of the `continue` statement must be within the same `finally` block; otherwise a compile-time error occurs.

A `continue` statement is executed as follows:

- If the `continue` statement exits one or more `try` blocks with associated `finally` blocks, control is initially transferred to the `finally` block of the innermost `try` statement. When and if control reaches the end point of a `finally` block, control is transferred to the `finally` block of the next enclosing `try` statement. This process is repeated until the `finally` blocks of all intervening `try` statements have been executed.
- Control is transferred to the target of the `continue` statement.

Because a `continue` statement unconditionally transfers control elsewhere, the end point of a `continue` statement is never reachable.

The `goto` statement

The `goto` statement transfers control to a statement that is marked by a label.

```
goto_statement
: 'goto' identifier ';'
| 'goto' 'case' constant_expression ';'
| 'goto' 'default' ';'
;
```

The target of a `goto` *identifier* statement is the labeled statement with the given label. If a label with the given

name does not exist in the current function member, or if the `goto` statement is not within the scope of the label, a compile-time error occurs. This rule permits the use of a `goto` statement to transfer control out of a nested scope, but not into a nested scope. In the example

```
using System;

class Test
{
    static void Main(string[] args) {
        string[,] table = {
            {"Red", "Blue", "Green"},
            {"Monday", "Wednesday", "Friday"}
        };

        foreach (string str in args) {
            int row, colm;
            for (row = 0; row <= 1; ++row)
                for (colm = 0; colm <= 2; ++colm)
                    if (str == table[row,colm])
                        goto done;

            Console.WriteLine("{0} not found", str);
            continue;
        done:
            Console.WriteLine("Found {0} at [{1}][{2}]", str, row, colm);
        }
    }
}
```

a `goto` statement is used to transfer control out of a nested scope.

The target of a `goto case` statement is the statement list in the immediately enclosing `switch` statement ([The switch statement](#)), which contains a `case` label with the given constant value. If the `goto case` statement is not enclosed by a `switch` statement, if the *constant expression* is not implicitly convertible ([Implicit conversions](#)) to the governing type of the nearest enclosing `switch` statement, or if the nearest enclosing `switch` statement does not contain a `case` label with the given constant value, a compile-time error occurs.

The target of a `goto default` statement is the statement list in the immediately enclosing `switch` statement ([The switch statement](#)), which contains a `default` label. If the `goto default` statement is not enclosed by a `switch` statement, or if the nearest enclosing `switch` statement does not contain a `default` label, a compile-time error occurs.

A `goto` statement cannot exit a `finally` block ([The try statement](#)). When a `goto` statement occurs within a `finally` block, the target of the `goto` statement must be within the same `finally` block, or otherwise a compile-time error occurs.

A `goto` statement is executed as follows:

- If the `goto` statement exits one or more `try` blocks with associated `finally` blocks, control is initially transferred to the `finally` block of the innermost `try` statement. When and if control reaches the end point of a `finally` block, control is transferred to the `finally` block of the next enclosing `try` statement. This process is repeated until the `finally` blocks of all intervening `try` statements have been executed.
- Control is transferred to the target of the `goto` statement.

Because a `goto` statement unconditionally transfers control elsewhere, the end point of a `goto` statement is never reachable.

The return statement

The `return` statement returns control to the current caller of the function in which the `return` statement

appears.

```
return_statement
: 'return' expression? ';'
;
```

A `return` statement with no expression can be used only in a function member that does not compute a value, that is, a method with the result type ([Method body](#)) `void`, the `set` accessor of a property or indexer, the `add` and `remove` accessors of an event, an instance constructor, a static constructor, or a destructor.

A `return` statement with an expression can only be used in a function member that computes a value, that is, a method with a non-void result type, the `get` accessor of a property or indexer, or a user-defined operator. An implicit conversion ([Implicit conversions](#)) must exist from the type of the expression to the return type of the containing function member.

Return statements can also be used in the body of anonymous function expressions ([Anonymous function expressions](#)), and participate in determining which conversions exist for those functions.

It is a compile-time error for a `return` statement to appear in a `finally` block ([The try statement](#)).

A `return` statement is executed as follows:

- If the `return` statement specifies an expression, the expression is evaluated and the resulting value is converted to the return type of the containing function by an implicit conversion. The result of the conversion becomes the result value produced by the function.
- If the `return` statement is enclosed by one or more `try` or `catch` blocks with associated `finally` blocks, control is initially transferred to the `finally` block of the innermost `try` statement. When and if control reaches the end point of a `finally` block, control is transferred to the `finally` block of the next enclosing `try` statement. This process is repeated until the `finally` blocks of all enclosing `try` statements have been executed.
- If the containing function is not an async function, control is returned to the caller of the containing function along with the result value, if any.
- If the containing function is an async function, control is returned to the current caller, and the result value, if any, is recorded in the return task as described in ([Enumerator interfaces](#)).

Because a `return` statement unconditionally transfers control elsewhere, the end point of a `return` statement is never reachable.

The throw statement

The `throw` statement throws an exception.

```
throw_statement
: 'throw' expression? ';'
;
```

A `throw` statement with an expression throws the value produced by evaluating the expression. The expression must denote a value of the class type `System.Exception`, of a class type that derives from `System.Exception` or of a type parameter type that has `System.Exception` (or a subclass thereof) as its effective base class. If evaluation of the expression produces `null`, a `System.NullReferenceException` is thrown instead.

A `throw` statement with no expression can be used only in a `catch` block, in which case that statement re-throws the exception that is currently being handled by that `catch` block.

Because a `throw` statement unconditionally transfers control elsewhere, the end point of a `throw` statement is never reachable.

When an exception is thrown, control is transferred to the first `catch` clause in an enclosing `try` statement that can handle the exception. The process that takes place from the point of the exception being thrown to the point of transferring control to a suitable exception handler is known as *exception propagation*. Propagation of an exception consists of repeatedly evaluating the following steps until a `catch` clause that matches the exception is found. In this description, the *throw point* is initially the location at which the exception is thrown.

- In the current function member, each `try` statement that encloses the throw point is examined. For each statement `S`, starting with the innermost `try` statement and ending with the outermost `try` statement, the following steps are evaluated:
 - If the `try` block of `S` encloses the throw point and if `S` has one or more `catch` clauses, the `catch` clauses are examined in order of appearance to locate a suitable handler for the exception, according to the rules specified in Section [The try statement](#). If a matching `catch` clause is located, the exception propagation is completed by transferring control to the block of that `catch` clause.
 - Otherwise, if the `try` block or a `catch` block of `S` encloses the throw point and if `S` has a `finally` block, control is transferred to the `finally` block. If the `finally` block throws another exception, processing of the current exception is terminated. Otherwise, when control reaches the end point of the `finally` block, processing of the current exception is continued.
- If an exception handler was not located in the current function invocation, the function invocation is terminated, and one of the following occurs:
 - If the current function is non-async, the steps above are repeated for the caller of the function with a throw point corresponding to the statement from which the function member was invoked.
 - If the current function is async and task-returning, the exception is recorded in the return task, which is put into a faulted or cancelled state as described in [Enumerator interfaces](#).
 - If the current function is async and void-returning, the synchronization context of the current thread is notified as described in [Enumerable interfaces](#).
- If the exception processing terminates all function member invocations in the current thread, indicating that the thread has no handler for the exception, then the thread is itself terminated. The impact of such termination is implementation-defined.

The try statement

The `try` statement provides a mechanism for catching exceptions that occur during execution of a block. Furthermore, the `try` statement provides the ability to specify a block of code that is always executed when control leaves the `try` statement.

```

try_statement
    : 'try' block catch_clause+
    | 'try' block finally_clause
    | 'try' block catch_clause+ finally_clause
    ;

catch_clause
    : 'catch' exception_specifier? exception_filter? block
    ;

exception_specifier
    : '(' type identifier? ')'
    ;

exception_filter
    : 'when' '(' expression ')'
    ;

finally_clause
    : 'finally' block
    ;

```

There are three possible forms of `try` statements:

- A `try` block followed by one or more `catch` blocks.
- A `try` block followed by a `finally` block.
- A `try` block followed by one or more `catch` blocks followed by a `finally` block.

When a `catch` clause specifies an *exception_specifier*, the type must be `System.Exception`, a type that derives from `System.Exception` or a type parameter type that has `System.Exception` (or a subclass thereof) as its effective base class.

When a `catch` clause specifies both an *exception_specifier* with an *identifier*, an **exception variable** of the given name and type is declared. The exception variable corresponds to a local variable with a scope that extends over the `catch` clause. During execution of the *exception_filter* and *block*, the exception variable represents the exception currently being handled. For purposes of definite assignment checking, the exception variable is considered definitely assigned in its entire scope.

Unless a `catch` clause includes an exception variable name, it is impossible to access the exception object in the filter and `catch` block.

A `catch` clause that does not specify an *exception_specifier* is called a general `catch` clause.

Some programming languages may support exceptions that are not representable as an object derived from `System.Exception`, although such exceptions could never be generated by C# code. A general `catch` clause may be used to catch such exceptions. Thus, a general `catch` clause is semantically different from one that specifies the type `System.Exception`, in that the former may also catch exceptions from other languages.

In order to locate a handler for an exception, `catch` clauses are examined in lexical order. If a `catch` clause specifies a type but no exception filter, it is a compile-time error for a later `catch` clause in the same `try` statement to specify a type that is the same as, or is derived from, that type. If a `catch` clause specifies no type and no filter, it must be the last `catch` clause for that `try` statement.

Within a `catch` block, a `throw` statement ([The throw statement](#)) with no expression can be used to re-throw the exception that was caught by the `catch` block. Assignments to an exception variable do not alter the exception that is re-thrown.

In the example

```

using System;

class Test
{
    static void F() {
        try {
            G();
        }
        catch (Exception e) {
            Console.WriteLine("Exception in F: " + e.Message);
            e = new Exception("F");
            throw;           // re-throw
        }
    }

    static void G() {
        throw new Exception("G");
    }

    static void Main() {
        try {
            F();
        }
        catch (Exception e) {
            Console.WriteLine("Exception in Main: " + e.Message);
        }
    }
}

```

the method `F` catches an exception, writes some diagnostic information to the console, alters the exception variable, and re-throws the exception. The exception that is re-thrown is the original exception, so the output produced is:

```

Exception in F: G
Exception in Main: G

```

If the first catch block had thrown `e` instead of rethrowing the current exception, the output produced would be as follows:

```

Exception in F: G
Exception in Main: F

```

It is a compile-time error for a `break`, `continue`, or `goto` statement to transfer control out of a `finally` block. When a `break`, `continue`, or `goto` statement occurs in a `finally` block, the target of the statement must be within the same `finally` block, or otherwise a compile-time error occurs.

It is a compile-time error for a `return` statement to occur in a `finally` block.

A `try` statement is executed as follows:

- Control is transferred to the `try` block.
- When and if control reaches the end point of the `try` block:
 - If the `try` statement has a `finally` block, the `finally` block is executed.
 - Control is transferred to the end point of the `try` statement.
- If an exception is propagated to the `try` statement during execution of the `try` block:
 - The `catch` clauses, if any, are examined in order of appearance to locate a suitable handler for the

exception. If a `catch` clause does not specify a type, or specifies the exception type or a base type of the exception type:

- If the `catch` clause declares an exception variable, the exception object is assigned to the exception variable.
- If the `catch` clause declares an exception filter, the filter is evaluated. If it evaluates to `false`, the catch clause is not a match, and the search continues through any subsequent `catch` clauses for a suitable handler.
- Otherwise, the `catch` clause is considered a match, and control is transferred to the matching `catch` block.
- When and if control reaches the end point of the `catch` block:
 - If the `try` statement has a `finally` block, the `finally` block is executed.
 - Control is transferred to the end point of the `try` statement.
- If an exception is propagated to the `try` statement during execution of the `catch` block:
 - If the `try` statement has a `finally` block, the `finally` block is executed.
 - The exception is propagated to the next enclosing `try` statement.
- If the `try` statement has no `catch` clauses or if no `catch` clause matches the exception:
 - If the `try` statement has a `finally` block, the `finally` block is executed.
 - The exception is propagated to the next enclosing `try` statement.

The statements of a `finally` block are always executed when control leaves a `try` statement. This is true whether the control transfer occurs as a result of normal execution, as a result of executing a `break`, `continue`, `goto`, or `return` statement, or as a result of propagating an exception out of the `try` statement.

If an exception is thrown during execution of a `finally` block, and is not caught within the same finally block, the exception is propagated to the next enclosing `try` statement. If another exception was in the process of being propagated, that exception is lost. The process of propagating an exception is discussed further in the description of the `throw` statement ([The throw statement](#)).

The `try` block of a `try` statement is reachable if the `try` statement is reachable.

A `catch` block of a `try` statement is reachable if the `try` statement is reachable.

The `finally` block of a `try` statement is reachable if the `try` statement is reachable.

The end point of a `try` statement is reachable if both of the following are true:

- The end point of the `try` block is reachable or the end point of at least one `catch` block is reachable.
- If a `finally` block is present, the end point of the `finally` block is reachable.

The checked and unchecked statements

The `checked` and `unchecked` statements are used to control the *overflow checking context* for integral-type arithmetic operations and conversions.

```
checked_statement
: 'checked' block
;

unchecked_statement
: 'unchecked' block
;
```

The `checked` statement causes all expressions in the *block* to be evaluated in a checked context, and the

`unchecked` statement causes all expressions in the *block* to be evaluated in an unchecked context.

The `checked` and `unchecked` statements are precisely equivalent to the `checked` and `unchecked` operators ([The checked and unchecked operators](#)), except that they operate on blocks instead of expressions.

The lock statement

The `lock` statement obtains the mutual-exclusion lock for a given object, executes a statement, and then releases the lock.

```
lock_statement
: 'lock' '(' expression ')' embedded_statement
;
```

The expression of a `lock` statement must denote a value of a type known to be a *reference_type*. No implicit boxing conversion ([Boxing conversions](#)) is ever performed for the expression of a `lock` statement, and thus it is a compile-time error for the expression to denote a value of a *value_type*.

A `lock` statement of the form

```
lock (x) ...
```

where `x` is an expression of a *reference_type*, is precisely equivalent to

```
bool __lockWasTaken = false;
try {
    System.Threading.Monitor.Enter(x, ref __lockWasTaken);
    ...
}
finally {
    if (__lockWasTaken) System.Threading.Monitor.Exit(x);
}
```

except that `x` is only evaluated once.

While a mutual-exclusion lock is held, code executing in the same execution thread can also obtain and release the lock. However, code executing in other threads is blocked from obtaining the lock until the lock is released.

Locking `System.Type` objects in order to synchronize access to static data is not recommended. Other code might lock on the same type, which can result in deadlock. A better approach is to synchronize access to static data by locking a private static object. For example:


```

class Cache
{
    private static readonly object synchronizationObject = new object();

    public static void Add(object x) {
        lock (Cache.synchronizationObject) {
            ...
        }
    }

    public static void Remove(object x) {
        lock (Cache.synchronizationObject) {
            ...
        }
    }
}

```

The using statement

The `using` statement obtains one or more resources, executes a statement, and then disposes of the resource.

```

using_statement
: 'using' '(' resource_acquisition ')' embedded_statement
;

resource_acquisition
: local_variable_declaration
| expression
;

```

A **resource** is a class or struct that implements `System.IDisposable`, which includes a single parameterless method named `Dispose`. Code that is using a resource can call `Dispose` to indicate that the resource is no longer needed. If `Dispose` is not called, then automatic disposal eventually occurs as a consequence of garbage collection.

If the form of *resource_acquisition* is *local_variable_declaration* then the type of the *local_variable_declaration* must be either `dynamic` or a type that can be implicitly converted to `System.IDisposable`. If the form of *resource_acquisition* is *expression* then this expression must be implicitly convertible to `System.IDisposable`.

Local variables declared in a *resource_acquisition* are read-only, and must include an initializer. A compile-time error occurs if the embedded statement attempts to modify these local variables (via assignment or the `++` and `--` operators), take the address of them, or pass them as `ref` or `out` parameters.

A `using` statement is translated into three parts: acquisition, usage, and disposal. Usage of the resource is implicitly enclosed in a `try` statement that includes a `finally` clause. This `finally` clause disposes of the resource. If a `null` resource is acquired, then no call to `Dispose` is made, and no exception is thrown. If the resource is of type `dynamic` it is dynamically converted through an implicit dynamic conversion ([Implicit dynamic conversions](#)) to `IDisposable` during acquisition in order to ensure that the conversion is successful before the usage and disposal.

A `using` statement of the form

```
using (ResourceType resource = expression) statement
```

corresponds to one of three possible expansions. When `ResourceType` is a non-nullable value type, the expansion is

```

{
    ResourceType resource = expression;
    try {
        statement;
    }
    finally {
        ((IDisposable)resource).Dispose();
    }
}

```

Otherwise, when `ResourceType` is a nullable value type or a reference type other than `dynamic`, the expansion is

```

{
    ResourceType resource = expression;
    try {
        statement;
    }
    finally {
        if (resource != null) ((IDisposable)resource).Dispose();
    }
}

```

Otherwise, when `ResourceType` is `dynamic`, the expansion is

```

{
    ResourceType resource = expression;
    IDisposable d = (IDisposable)resource;
    try {
        statement;
    }
    finally {
        if (d != null) d.Dispose();
    }
}

```

In either expansion, the `resource` variable is read-only in the embedded statement, and the `d` variable is inaccessible in, and invisible to, the embedded statement.

An implementation is permitted to implement a given using-statement differently, e.g. for performance reasons, as long as the behavior is consistent with the above expansion.

A `using` statement of the form

```
using (expression) statement
```

has the same three possible expansions. In this case `ResourceType` is implicitly the compile-time type of the `expression`, if it has one. Otherwise the interface `IDisposable` itself is used as the `ResourceType`. The `resource` variable is inaccessible in, and invisible to, the embedded statement.

When a *resource_acquisition* takes the form of a *local_variable_declaration*, it is possible to acquire multiple resources of a given type. A `using` statement of the form

```
using (ResourceType r1 = e1, r2 = e2, ..., rN = eN) statement
```

is precisely equivalent to a sequence of nested `using` statements:

```

using (ResourceType r1 = e1)
    using (ResourceType r2 = e2)
        ...
            using (ResourceType rN = eN)
                statement

```

The example below creates a file named `log.txt` and writes two lines of text to the file. The example then opens that same file for reading and copies the contained lines of text to the console.

```

using System;
using System.IO;

class Test
{
    static void Main() {
        using (TextWriter w = File.CreateText("log.txt")) {
            w.WriteLine("This is line one");
            w.WriteLine("This is line two");
        }

        using (TextReader r = File.OpenText("log.txt")) {
            string s;
            while ((s = r.ReadLine()) != null) {
                Console.WriteLine(s);
            }
        }
    }
}

```

Since the `TextWriter` and `TextReader` classes implement the `IDisposable` interface, the example can use `using` statements to ensure that the underlying file is properly closed following the write or read operations.

The yield statement

The `yield` statement is used in an iterator block ([Blocks](#)) to yield a value to the enumerator object ([Enumerator objects](#)) or enumerable object ([Enumerable objects](#)) of an iterator or to signal the end of the iteration.

```

yield_statement
: 'yield' 'return' expression ';'
| 'yield' 'break' ';'
;

```

`yield` is not a reserved word; it has special meaning only when used immediately before a `return` or `break` keyword. In other contexts, `yield` can be used as an identifier.

There are several restrictions on where a `yield` statement can appear, as described in the following.

- It is a compile-time error for a `yield` statement (of either form) to appear outside a *method_body*, *operator_body* or *accessor_body*
- It is a compile-time error for a `yield` statement (of either form) to appear inside an anonymous function.
- It is a compile-time error for a `yield` statement (of either form) to appear in the `finally` clause of a `try` statement.
- It is a compile-time error for a `yield return` statement to appear anywhere in a `try` statement that contains any `catch` clauses.

The following example shows some valid and invalid uses of `yield` statements.

```

delegate IEnumerable<int> D();

IEnumerator<int> GetEnumerator() {
    try {
        yield return 1;           // Ok
        yield break;             // Ok
    }
    finally {
        yield return 2;          // Error, yield in finally
        yield break;             // Error, yield in finally
    }

    try {
        yield return 3;          // Error, yield return in try...catch
        yield break;             // Ok
    }
    catch {
        yield return 4;          // Error, yield return in try...catch
        yield break;             // Ok
    }

    D d = delegate {
        yield return 5;          // Error, yield in an anonymous function
    };
}

int MyMethod() {
    yield return 1;              // Error, wrong return type for an iterator block
}

```

An implicit conversion ([Implicit conversions](#)) must exist from the type of the expression in the `yield return` statement to the yield type ([Yield type](#)) of the iterator.

A `yield return` statement is executed as follows:

- The expression given in the statement is evaluated, implicitly converted to the yield type, and assigned to the `Current` property of the enumerator object.
- Execution of the iterator block is suspended. If the `yield return` statement is within one or more `try` blocks, the associated `finally` blocks are not executed at this time.
- The `MoveNext` method of the enumerator object returns `true` to its caller, indicating that the enumerator object successfully advanced to the next item.

The next call to the enumerator object's `MoveNext` method resumes execution of the iterator block from where it was last suspended.

A `yield break` statement is executed as follows:

- If the `yield break` statement is enclosed by one or more `try` blocks with associated `finally` blocks, control is initially transferred to the `finally` block of the innermost `try` statement. When and if control reaches the end point of a `finally` block, control is transferred to the `finally` block of the next enclosing `try` statement. This process is repeated until the `finally` blocks of all enclosing `try` statements have been executed.
- Control is returned to the caller of the iterator block. This is either the `MoveNext` method or `Dispose` method of the enumerator object.

Because a `yield break` statement unconditionally transfers control elsewhere, the end point of a `yield break` statement is never reachable.

Namespaces

12/28/2021 • 18 minutes to read • [Edit Online](#)

C# programs are organized using namespaces. Namespaces are used both as an "internal" organization system for a program, and as an "external" organization system—a way of presenting program elements that are exposed to other programs.

Using directives ([Using directives](#)) are provided to facilitate the use of namespaces.

Compilation units

A *compilation_unit* defines the overall structure of a source file. A compilation unit consists of zero or more *using_directives* followed by zero or more *global_attributes* followed by zero or more *namespace_member_declarations*.

```
compilation_unit
: extern_alias_directive* using_directive* global_attributes? namespace_member_declaration*
;
```

A C# program consists of one or more compilation units, each contained in a separate source file. When a C# program is compiled, all of the compilation units are processed together. Thus, compilation units can depend on each other, possibly in a circular fashion.

The *using_directives* of a compilation unit affect the *global_attributes* and *namespace_member_declarations* of that compilation unit, but have no effect on other compilation units.

The *global_attributes* ([Attributes](#)) of a compilation unit permit the specification of attributes for the target assembly and module. Assemblies and modules act as physical containers for types. An assembly may consist of several physically separate modules.

The *namespace_member_declarations* of each compilation unit of a program contribute members to a single declaration space called the global namespace. For example:

File `A.cs`:

```
class A {}
```

File `B.cs`:

```
class B {}
```

The two compilation units contribute to the single global namespace, in this case declaring two classes with the fully qualified names `A` and `B`. Because the two compilation units contribute to the same declaration space, it would have been an error if each contained a declaration of a member with the same name.

Namespace declarations

A *namespace_declaration* consists of the keyword `namespace`, followed by a namespace name and body, optionally followed by a semicolon.

```

namespace_declaration
    : 'namespace' qualified_identifier namespace_body ';' '?'
    ;

qualified_identifier
    : identifier ('.' identifier)*
    ;

namespace_body
    : '{' extern_alias_directive* using_directive* namespace_member_declaration* '}'
    ;

```

A *namespace_declaration* may occur as a top-level declaration in a *compilation_unit* or as a member declaration within another *namespace_declaration*. When a *namespace_declaration* occurs as a top-level declaration in a *compilation_unit*, the namespace becomes a member of the global namespace. When a *namespace_declaration* occurs within another *namespace_declaration*, the inner namespace becomes a member of the outer namespace. In either case, the name of a namespace must be unique within the containing namespace.

Namespaces are implicitly `public` and the declaration of a namespace cannot include any access modifiers.

Within a *namespace_body*, the optional *using_directives* import the names of other namespaces, types and members, allowing them to be referenced directly instead of through qualified names. The optional *namespace_member_declarations* contribute members to the declaration space of the namespace. Note that all *using_directives* must appear before any member declarations.

The *qualified_identifier* of a *namespace_declaration* may be a single identifier or a sequence of identifiers separated by "." tokens. The latter form permits a program to define a nested namespace without lexically nesting several namespace declarations. For example,

```

namespace N1.N2
{
    class A {}

    class B {}
}

```

is semantically equivalent to

```

namespace N1
{
    namespace N2
    {
        class A {}

        class B {}
    }
}

```

Namespaces are open-ended, and two namespace declarations with the same fully qualified name contribute to the same declaration space ([Declarations](#)). In the example

```

namespace N1.N2
{
    class A {}
}

namespace N1.N2
{
    class B {}
}

```

the two namespace declarations above contribute to the same declaration space, in this case declaring two classes with the fully qualified names `N1.N2.A` and `N1.N2.B`. Because the two declarations contribute to the same declaration space, it would have been an error if each contained a declaration of a member with the same name.

Extern aliases

An *extern_alias_directive* introduces an identifier that serves as an alias for a namespace. The specification of the aliased namespace is external to the source code of the program and applies also to nested namespaces of the aliased namespace.

```

extern_alias_directive
: 'extern' 'alias' identifier ';'
;

```

The scope of an *extern_alias_directive* extends over the *using_directives*, *global_attributes* and *namespace_member_declarations* of its immediately containing compilation unit or namespace body.

Within a compilation unit or namespace body that contains an *extern_alias_directive*, the identifier introduced by the *extern_alias_directive* can be used to reference the aliased namespace. It is a compile-time error for the *identifier* to be the word `global`.

An *extern_alias_directive* makes an alias available within a particular compilation unit or namespace body, but it does not contribute any new members to the underlying declaration space. In other words, an *extern_alias_directive* is not transitive, but, rather, affects only the compilation unit or namespace body in which it occurs.

The following program declares and uses two extern aliases, `x` and `y`, each of which represent the root of a distinct namespace hierarchy:

```

extern alias X;
extern alias Y;

class Test
{
    X::N.A a;
    X::N.B b1;
    Y::N.B b2;
    Y::N.C c;
}

```

The program declares the existence of the extern aliases `x` and `y`, but the actual definitions of the aliases are external to the program. The identically named `N.B` classes can now be referenced as `x.N.B` and `y.N.B`, or, using the namespace alias qualifier, `x::N.B` and `y::N.B`. An error occurs if a program declares an extern alias for which no external definition is provided.

Using directives

Using directives facilitate the use of namespaces and types defined in other namespaces. Using directives impact the name resolution process of *namespace_or_type_names* ([Namespace and type names](#)) and *simple_names* ([Simple names](#)), but unlike declarations, using directives do not contribute new members to the underlying declaration spaces of the compilation units or namespaces within which they are used.

```
using_directive
: using_alias_directive
| using_namespace_directive
| using_static_directive
;
```

A *using_alias_directive* ([Using alias directives](#)) introduces an alias for a namespace or type.

A *using_namespace_directive* ([Using namespace directives](#)) imports the type members of a namespace.

A *using_static_directive* ([Using static directives](#)) imports the nested types and static members of a type.

The scope of a *using_directive* extends over the *namespace_member_declarations* of its immediately containing compilation unit or namespace body. The scope of a *using_directive* specifically does not include its peer *using_directives*. Thus, peer *using_directives* do not affect each other, and the order in which they are written is insignificant.

Using alias directives

A *using_alias_directive* introduces an identifier that serves as an alias for a namespace or type within the immediately enclosing compilation unit or namespace body.

```
using_alias_directive
: 'using' identifier '=' namespace_or_type_name ';'
;
```

Within member declarations in a compilation unit or namespace body that contains a *using_alias_directive*, the identifier introduced by the *using_alias_directive* can be used to reference the given namespace or type. For example:

```
namespace N1.N2
{
    class A {}
}

namespace N3
{
    using A = N1.N2.A;

    class B: A {}
}
```

Above, within member declarations in the `N3` namespace, `A` is an alias for `N1.N2.A`, and thus class `N3.B` derives from class `N1.N2.A`. The same effect can be obtained by creating an alias `R` for `N1.N2` and then referencing `R.A`:


```

namespace N3
{
    using R = N1.N2;

    class B: R.A {}
}

```

The *identifier* of a *using_alias_directive* must be unique within the declaration space of the compilation unit or namespace that immediately contains the *using_alias_directive*. For example:

```

namespace N3
{
    class A {}
}

namespace N3
{
    using A = N1.N2.A;      // Error, A already exists
}

```

Above, `N3` already contains a member `A`, so it is a compile-time error for a *using_alias_directive* to use that identifier. Likewise, it is a compile-time error for two or more *using_alias_directives* in the same compilation unit or namespace body to declare aliases by the same name.

A *using_alias_directive* makes an alias available within a particular compilation unit or namespace body, but it does not contribute any new members to the underlying declaration space. In other words, a *using_alias_directive* is not transitive but rather affects only the compilation unit or namespace body in which it occurs. In the example

```

namespace N3
{
    using R = N1.N2;
}

namespace N3
{
    class B: R.A {}          // Error, R unknown
}

```

the scope of the *using_alias_directive* that introduces `R` only extends to member declarations in the namespace body in which it is contained, so `R` is unknown in the second namespace declaration. However, placing the *using_alias_directive* in the containing compilation unit causes the alias to become available within both namespace declarations:

```

using R = N1.N2;

namespace N3
{
    class B: R.A {}
}

namespace N3
{
    class C: R.A {}
}

```

Just like regular members, names introduced by *using_alias_directives* are hidden by similarly named members in nested scopes. In the example

```

using R = N1.N2;

namespace N3
{
    class R {}

    class B: R.A {}          // Error, R has no member A
}

```

the reference to `R.A` in the declaration of `B` causes a compile-time error because `R` refers to `N3.R`, not `N1.N2`.

The order in which *using_alias_directives* are written has no significance, and resolution of the *namespace_or_type_name* referenced by a *using_alias_directive* is not affected by the *using_alias_directive* itself or by other *using_directives* in the immediately containing compilation unit or namespace body. In other words, the *namespace_or_type_name* of a *using_alias_directive* is resolved as if the immediately containing compilation unit or namespace body had no *using_directives*. A *using_alias_directive* may however be affected by *extern_alias_directives* in the immediately containing compilation unit or namespace body. In the example

```

namespace N1.N2 {}

namespace N3
{
    extern alias E;

    using R1 = E.N;          // OK

    using R2 = N1;           // OK

    using R3 = N1.N2;        // OK

    using R4 = R2.N2;        // Error, R2 unknown
}

```

the last *using_alias_directive* results in a compile-time error because it is not affected by the first *using_alias_directive*. The first *using_alias_directive* does not result in an error since the scope of the extern alias `E` includes the *using_alias_directive*.

A *using_alias_directive* can create an alias for any namespace or type, including the namespace within which it appears and any namespace or type nested within that namespace.

Accessing a namespace or type through an alias yields exactly the same result as accessing that namespace or type through its declared name. For example, given

```

namespace N1.N2
{
    class A {}
}

namespace N3
{
    using R1 = N1;
    using R2 = N1.N2;

    class B
    {
        N1.N2.A a;           // refers to N1.N2.A
        R1.N2.A b;           // refers to N1.N2.A
        R2.A c;              // refers to N1.N2.A
    }
}

```

the names `N1.N2.A`, `R1.N2.A`, and `R2.A` are equivalent and all refer to the class whose fully qualified name is `N1.N2.A`.

Using aliases can name a closed constructed type, but cannot name an unbound generic type declaration without supplying type arguments. For example:

```

namespace N1
{
    class A<T>
    {
        class B {}
    }
}

namespace N2
{
    using W = N1.A;           // Error, cannot name unbound generic type

    using X = N1.A.B;         // Error, cannot name unbound generic type

    using Y = N1.A<int>;      // Ok, can name closed constructed type

    using Z<T> = N1.A<T>;     // Error, using alias cannot have type parameters
}

```

Using namespace directives

A *using_namespace_directive* imports the types contained in a namespace into the immediately enclosing compilation unit or namespace body, enabling the identifier of each type to be used without qualification.

```

using_namespace_directive
: 'using' namespace_name ';'
;

```

Within member declarations in a compilation unit or namespace body that contains a *using_namespace_directive*, the types contained in the given namespace can be referenced directly. For example:

```

namespace N1.N2
{
    class A {}
}

namespace N3
{
    using N1.N2;

    class B: A {}
}

```

Above, within member declarations in the `N3` namespace, the type members of `N1.N2` are directly available, and thus class `N3.B` derives from class `N1.N2.A`.

A *using_namespace_directive* imports the types contained in the given namespace, but specifically does not import nested namespaces. In the example

```

namespace N1.N2
{
    class A {}
}

namespace N3
{
    using N1;

    class B: N2.A {}      // Error, N2 unknown
}

```

the *using_namespace_directive* imports the types contained in `N1`, but not the namespaces nested in `N1`. Thus, the reference to `N2.A` in the declaration of `B` results in a compile-time error because no members named `N2` are in scope.

Unlike a *using_alias_directive*, a *using_namespace_directive* may import types whose identifiers are already defined within the enclosing compilation unit or namespace body. In effect, names imported by a *using_namespace_directive* are hidden by similarly named members in the enclosing compilation unit or namespace body. For example:

```

namespace N1.N2
{
    class A {}

    class B {}
}

namespace N3
{
    using N1.N2;

    class A {}
}

```

Here, within member declarations in the `N3` namespace, `A` refers to `N3.A` rather than `N1.N2.A`.

When more than one namespace or type imported by *using_namespace_directives* or *using_static_directives* in the same compilation unit or namespace body contain types by the same name, references to that name as a *type_name* are considered ambiguous. In the example

```

namespace N1
{
    class A {}
}

namespace N2
{
    class A {}
}

namespace N3
{
    using N1;

    using N2;

    class B: A {}           // Error, A is ambiguous
}

```

both `N1` and `N2` contain a member `A`, and because `N3` imports both, referencing `A` in `N3` is a compile-time error. In this situation, the conflict can be resolved either through qualification of references to `A`, or by introducing a *using_alias_directive* that picks a particular `A`. For example:

```

namespace N3
{
    using N1;

    using N2;

    using A = N1.A;

    class B: A {}           // A means N1.A
}

```

Furthermore, when more than one namespace or type imported by *using_namespace_directives* or *using_static_directives* in the same compilation unit or namespace body contain types or members by the same name, references to that name as a *simple_name* are considered ambiguous. In the example

```

namespace N1
{
    class A {}
}

class C
{
    public static int A;
}

namespace N2
{
    using N1;
    using static C;

    class B
    {
        void M()
        {
            A a = new A();    // Ok, A is unambiguous as a type-name
            A.Equals(2);      // Error, A is ambiguous as a simple-name
        }
    }
}

```

`N1` contains a type member `A`, and `C` contains a static field `A`, and because `N2` imports both, referencing `A` as a *simple_name* is ambiguous and a compile-time error.

Like a *using_alias_directive*, a *using_namespace_directive* does not contribute any new members to the underlying declaration space of the compilation unit or namespace, but rather affects only the compilation unit or namespace body in which it appears.

The *namespace_name* referenced by a *using_namespace_directive* is resolved in the same way as the *namespace_or_type_name* referenced by a *using_alias_directive*. Thus, *using_namespace_directives* in the same compilation unit or namespace body do not affect each other and can be written in any order.

Using static directives

A *using_static_directive* imports the nested types and static members contained directly in a type declaration into the immediately enclosing compilation unit or namespace body, enabling the identifier of each member and type to be used without qualification.

```

using_static_directive
    : 'using' 'static' type_name ';'
    ;

```

Within member declarations in a compilation unit or namespace body that contains a *using_static_directive*, the accessible nested types and static members (except extension methods) contained directly in the declaration of the given type can be referenced directly. For example:

```

namespace N1
{
    class A
    {
        public class B{}
        public static B M(){ return new B(); }
    }
}

namespace N2
{
    using static N1.A;
    class C
    {
        void N() { B b = M(); }
    }
}

```

Above, within member declarations in the `N2` namespace, the static members and nested types of `N1.A` are directly available, and thus the method `N` is able to reference both the `B` and `M` members of `N1.A`.

A *using_static_directive* specifically does not import extension methods directly as static methods, but makes them available for extension method invocation ([Extension method invocations](#)). In the example

```

namespace N1
{
    static class A
    {
        public static void M(this string s){}
    }
}

namespace N2
{
    using static N1.A;

    class B
    {
        void N()
        {
            M("A");      // Error, M unknown
            "B".M();      // Ok, M known as extension method
            N1.A.M("C"); // Ok, fully qualified
        }
    }
}

```

the *using_static_directive* imports the extension method `M` contained in `N1.A`, but only as an extension method. Thus, the first reference to `M` in the body of `B.N` results in a compile-time error because no members named `M` are in scope.

A *using_static_directive* only imports members and types declared directly in the given type, not members and types declared in base classes.

TODO: Example

Ambiguities between multiple *using_namespace_directives* and *using_static_directives* are discussed in [Using namespace directives](#).

Namespace members

A *namespace_member_declaration* is either a *namespace_declaration* ([Namespace declarations](#)) or a *type_declaration* ([Type declarations](#)).

```
namespace_member_declaration
: namespace_declaration
| type_declaration
;
```

A compilation unit or a namespace body can contain *namespace_member_declarations*, and such declarations contribute new members to the underlying declaration space of the containing compilation unit or namespace body.

Type declarations

A *type_declaration* is a *class_declaration* ([Class declarations](#)), a *struct_declaration* ([Struct declarations](#)), an *interface_declaration* ([Interface declarations](#)), an *enum_declaration* ([Enum declarations](#)), or a *delegate_declaration* ([Delegate declarations](#)).

```
type_declaration
: class_declaration
| struct_declaration
| interface_declaration
| enum_declaration
| delegate_declaration
;
```

A *type_declaration* can occur as a top-level declaration in a compilation unit or as a member declaration within a namespace, class, or struct.

When a type declaration for a type `T` occurs as a top-level declaration in a compilation unit, the fully qualified name of the newly declared type is simply `T`. When a type declaration for a type `T` occurs within a namespace, class, or struct, the fully qualified name of the newly declared type is `N.T`, where `N` is the fully qualified name of the containing namespace, class, or struct.

A type declared within a class or struct is called a nested type ([Nested types](#)).

The permitted access modifiers and the default access for a type declaration depend on the context in which the declaration takes place ([Declared accessibility](#)):

- Types declared in compilation units or namespaces can have `public` or `internal` access. The default is `internal` access.
- Types declared in classes can have `public`, `protected internal`, `protected`, `internal`, or `private` access. The default is `private` access.
- Types declared in structs can have `public`, `internal`, or `private` access. The default is `private` access.

Namespace alias qualifiers

The *namespace alias qualifier* `::` makes it possible to guarantee that type name lookups are unaffected by the introduction of new types and members. The namespace alias qualifier always appears between two identifiers referred to as the left-hand and right-hand identifiers. Unlike the regular `.` qualifier, the left-hand identifier of the `::` qualifier is looked up only as an extern or using alias.

A *qualified_alias_member* is defined as follows:


```
qualified_alias_member
: identifier '::' identifier type_argument_list?
;
```

A *qualified_alias_member* can be used as a *namespace_or_type_name* ([Namespace and type names](#)) or as the left operand in a *member_access* ([Member access](#)).

A *qualified_alias_member* has one of two forms:

- $N::I\langle A_1, \dots, A_k \rangle$, where N and I represent identifiers, and $\langle A_1, \dots, A_k \rangle$ is a type argument list. (k is always at least one.)
- $N::I$, where N and I represent identifiers. (In this case, k is considered to be zero.)

Using this notation, the meaning of a *qualified_alias_member* is determined as follows:

- If N is the identifier `global`, then the global namespace is searched for I :
 - If the global namespace contains a namespace named I and k is zero, then the *qualified_alias_member* refers to that namespace.
 - Otherwise, if the global namespace contains a non-generic type named I and k is zero, then the *qualified_alias_member* refers to that type.
 - Otherwise, if the global namespace contains a type named I that has k type parameters, then the *qualified_alias_member* refers to that type constructed with the given type arguments.
 - Otherwise, the *qualified_alias_member* is undefined and a compile-time error occurs.
- Otherwise, starting with the namespace declaration ([Namespace declarations](#)) immediately containing the *qualified_alias_member* (if any), continuing with each enclosing namespace declaration (if any), and ending with the compilation unit containing the *qualified_alias_member*, the following steps are evaluated until an entity is located:
 - If the namespace declaration or compilation unit contains a *using_alias_directive* that associates N with a type, then the *qualified_alias_member* is undefined and a compile-time error occurs.
 - Otherwise, if the namespace declaration or compilation unit contains an *extern_alias_directive* or *using_alias_directive* that associates N with a namespace, then:
 - If the namespace associated with N contains a namespace named I and k is zero, then the *qualified_alias_member* refers to that namespace.
 - Otherwise, if the namespace associated with N contains a non-generic type named I and k is zero, then the *qualified_alias_member* refers to that type.
 - Otherwise, if the namespace associated with N contains a type named I that has k type parameters, then the *qualified_alias_member* refers to that type constructed with the given type arguments.
 - Otherwise, the *qualified_alias_member* is undefined and a compile-time error occurs.
- Otherwise, the *qualified_alias_member* is undefined and a compile-time error occurs.

Note that using the namespace alias qualifier with an alias that references a type causes a compile-time error. Also note that if the identifier N is `global`, then lookup is performed in the global namespace, even if there is a using alias associating `global` with a type or namespace.

Uniqueness of aliases

Each compilation unit and namespace body has a separate declaration space for extern aliases and using aliases. Thus, while the name of an extern alias or using alias must be unique within the set of extern aliases and using aliases declared in the immediately containing compilation unit or namespace body, an alias is permitted to have the same name as a type or namespace as long as it is used only with the `::` qualifier.

In the example

```
namespace N
{
    public class A {}

    public class B {}
}

namespace N
{
    using A = System.IO;

    class X
    {
        A.Stream s1;           // Error, A is ambiguous

        A::Stream s2;         // Ok
    }
}
```

the name `A` has two possible meanings in the second namespace body because both the class `A` and the using alias `A` are in scope. For this reason, use of `A` in the qualified name `A.Stream` is ambiguous and causes a compile-time error to occur. However, use of `A` with the `::` qualifier is not an error because `A` is looked up only as a namespace alias.

Classes

12/28/2021 • 170 minutes to read • [Edit Online](#)

A class is a data structure that may contain data members (constants and fields), function members (methods, properties, events, indexers, operators, instance constructors, destructors and static constructors), and nested types. Class types support inheritance, a mechanism whereby a derived class can extend and specialize a base class.

Class declarations

A *class_declaration* is a *type_declaration* ([Type declarations](#)) that declares a new class.

```
class_declaration
: attributes? class_modifier* 'partial'? 'class' identifier type_parameter_list?
  class_base? type_parameter_constraints_clause* class_body ';'?
```

A *class_declaration* consists of an optional set of *attributes* ([Attributes](#)), followed by an optional set of *class_modifiers* ([Class modifiers](#)), followed by an optional `partial` modifier, followed by the keyword `class` and an *identifier* that names the class, followed by an optional *type_parameter_list* ([Type parameters](#)), followed by an optional *class_base* specification ([Class base specification](#)), followed by an optional set of *type_parameter_constraints_clauses* ([Type parameter constraints](#)), followed by a *class_body* ([Class body](#)), optionally followed by a semicolon.

A class declaration cannot supply *type_parameter_constraints_clauses* unless it also supplies a *type_parameter_list*.

A class declaration that supplies a *type_parameter_list* is a **generic class declaration**. Additionally, any class nested inside a generic class declaration or a generic struct declaration is itself a generic class declaration, since type parameters for the containing type must be supplied to create a constructed type.

Class modifiers

A *class_declaration* may optionally include a sequence of class modifiers:

```
class_modifier
: 'new'
| 'public'
| 'protected'
| 'internal'
| 'private'
| 'abstract'
| 'sealed'
| 'static'
| class_modifier_unsafe
```

It is a compile-time error for the same modifier to appear multiple times in a class declaration.

The `new` modifier is permitted on nested classes. It specifies that the class hides an inherited member by the same name, as described in [The new modifier](#). It is a compile-time error for the `new` modifier to appear on a class declaration that is not a nested class declaration.

The `public`, `protected`, `internal`, and `private` modifiers control the accessibility of the class. Depending on

the context in which the class declaration occurs, some of these modifiers may not be permitted ([Declared accessibility](#)).

The `abstract`, `sealed` and `static` modifiers are discussed in the following sections.

Abstract classes

The `abstract` modifier is used to indicate that a class is incomplete and that it is intended to be used only as a base class. An abstract class differs from a non-abstract class in the following ways:

- An abstract class cannot be instantiated directly, and it is a compile-time error to use the `new` operator on an abstract class. While it is possible to have variables and values whose compile-time types are abstract, such variables and values will necessarily either be `null` or contain references to instances of non-abstract classes derived from the abstract types.
- An abstract class is permitted (but not required) to contain abstract members.
- An abstract class cannot be sealed.

When a non-abstract class is derived from an abstract class, the non-abstract class must include actual implementations of all inherited abstract members, thereby overriding those abstract members. In the example

```
abstract class A
{
    public abstract void F();
}

abstract class B: A
{
    public void G() {}
}

class C: B
{
    public override void F() {
        // actual implementation of F
    }
}
```

the abstract class `A` introduces an abstract method `F`. Class `B` introduces an additional method `G`, but since it doesn't provide an implementation of `F`, `B` must also be declared abstract. Class `C` overrides `F` and provides an actual implementation. Since there are no abstract members in `C`, `C` is permitted (but not required) to be non-abstract.

Sealed classes

The `sealed` modifier is used to prevent derivation from a class. A compile-time error occurs if a sealed class is specified as the base class of another class.

A sealed class cannot also be an abstract class.

The `sealed` modifier is primarily used to prevent unintended derivation, but it also enables certain run-time optimizations. In particular, because a sealed class is known to never have any derived classes, it is possible to transform virtual function member invocations on sealed class instances into non-virtual invocations.

Static classes

The `static` modifier is used to mark the class being declared as a *static class*. A static class cannot be instantiated, cannot be used as a type and can contain only static members. Only a static class can contain declarations of extension methods ([Extension methods](#)).

A static class declaration is subject to the following restrictions:

- A static class may not include a `sealed` or `abstract` modifier. Note, however, that since a static class cannot

be instantiated or derived from, it behaves as if it was both sealed and abstract.

- A static class may not include a *class_base* specification ([Class base specification](#)) and cannot explicitly specify a base class or a list of implemented interfaces. A static class implicitly inherits from type `object`.
- A static class can only contain static members ([Static and instance members](#)). Note that constants and nested types are classified as static members.
- A static class cannot have members with `protected` or `protected internal` declared accessibility.

It is a compile-time error to violate any of these restrictions.

A static class has no instance constructors. It is not possible to declare an instance constructor in a static class, and no default instance constructor ([Default constructors](#)) is provided for a static class.

The members of a static class are not automatically static, and the member declarations must explicitly include a `static` modifier (except for constants and nested types). When a class is nested within a static outer class, the nested class is not a static class unless it explicitly includes a `static` modifier.

Referencing static class types

A *namespace_or_type_name* ([Namespace and type names](#)) is permitted to reference a static class if

- The *namespace_or_type_name* is the `T` in a *namespace_or_type_name* of the form `T.I`, or
- The *namespace_or_type_name* is the `T` in a *typeof_expression* ([Argument lists](#)¹) of the form `typeof(T)`.

A *primary_expression* ([Function members](#)) is permitted to reference a static class if

- The *primary_expression* is the `E` in a *member_access* ([Compile-time checking of dynamic overload resolution](#)) of the form `E.I`.

In any other context it is a compile-time error to reference a static class. For example, it is an error for a static class to be used as a base class, a constituent type ([Nested types](#)) of a member, a generic type argument, or a type parameter constraint. Likewise, a static class cannot be used in an array type, a pointer type, a `new` expression, a cast expression, an `is` expression, an `as` expression, a `sizeof` expression, or a default value expression.

Partial modifier

The `partial` modifier is used to indicate that this *class_declaration* is a partial type declaration. Multiple partial type declarations with the same name within an enclosing namespace or type declaration combine to form one type declaration, following the rules specified in [Partial types](#).

Having the declaration of a class distributed over separate segments of program text can be useful if these segments are produced or maintained in different contexts. For instance, one part of a class declaration may be machine generated, whereas the other is manually authored. Textual separation of the two prevents updates by one from conflicting with updates by the other.

Type parameters

A type parameter is a simple identifier that denotes a placeholder for a type argument supplied to create a constructed type. A type parameter is a formal placeholder for a type that will be supplied later. By contrast, a type argument ([Type arguments](#)) is the actual type that is substituted for the type parameter when a constructed type is created.

```

type_parameter_list
: '<' type_parameters '>'
;

type_parameters
: attributes? type_parameter
| type_parameters ',' attributes? type_parameter
;

type_parameter
: identifier
;

```

Each type parameter in a class declaration defines a name in the declaration space ([Declarations](#)) of that class. Thus, it cannot have the same name as another type parameter or a member declared in that class. A type parameter cannot have the same name as the type itself.

Class base specification

A class declaration may include a *class_base* specification, which defines the direct base class of the class and the interfaces ([Interfaces](#)) directly implemented by the class.

```

class_base
: ':' class_type
| ':' interface_type_list
| ':' class_type ',' interface_type_list
;

interface_type_list
: interface_type (',' interface_type)*
;

```

The base class specified in a class declaration can be a constructed class type ([Constructed types](#)). A base class cannot be a type parameter on its own, though it can involve the type parameters that are in scope.

```

class Extend<V>: V {}           // Error, type parameter used as base class

```

Base classes

When a *class_type* is included in the *class_base*, it specifies the direct base class of the class being declared. If a class declaration has no *class_base*, or if the *class_base* lists only interface types, the direct base class is assumed to be `object`. A class inherits members from its direct base class, as described in [Inheritance](#).

In the example

```

class A {}

class B: A {}

```

class `A` is said to be the direct base class of `B`, and `B` is said to be derived from `A`. Since `A` does not explicitly specify a direct base class, its direct base class is implicitly `object`.

For a constructed class type, if a base class is specified in the generic class declaration, the base class of the constructed type is obtained by substituting, for each *type_parameter* in the base class declaration, the corresponding *type_argument* of the constructed type. Given the generic class declarations

```
class B<U,V> {...}

class G<T>: B<string,T[]> {...}
```

the base class of the constructed type `G<int>` would be `B<string,int[]>`.

The direct base class of a class type must be at least as accessible as the class type itself ([Accessibility domains](#)). For example, it is a compile-time error for a `public` class to derive from a `private` or `internal` class.

The direct base class of a class type must not be any of the following types: `System.Array`, `System.Delegate`, `System.MulticastDelegate`, `System.Enum`, or `System.ValueType`. Furthermore, a generic class declaration cannot use `System.Attribute` as a direct or indirect base class.

While determining the meaning of the direct base class specification `A` of a class `B`, the direct base class of `B` is temporarily assumed to be `object`. Intuitively this ensures that the meaning of a base class specification cannot recursively depend on itself. The example:

```
class A<T> {
    public class B {}
}

class C : A<C.B> {}
```

is in error since in the base class specification `A<C.B>` the direct base class of `C` is considered to be `object`, and hence (by the rules of [Namespace and type names](#)) `C` is not considered to have a member `B`.

The base classes of a class type are the direct base class and its base classes. In other words, the set of base classes is the transitive closure of the direct base class relationship. Referring to the example above, the base classes of `B` are `A` and `object`. In the example

```
class A {...}

class B<T>: A {...}

class C<T>: B<IComparable<T>> {...}

class D<T>: C<T[]> {...}
```

the base classes of `D<int>` are `C<int[]>`, `B<IComparable<int[]>>`, `A`, and `object`.

Except for class `object`, every class type has exactly one direct base class. The `object` class has no direct base class and is the ultimate base class of all other classes.

When a class `B` derives from a class `A`, it is a compile-time error for `A` to depend on `B`. A class **directly depends on** its direct base class (if any) and **directly depends on** the class within which it is immediately nested (if any). Given this definition, the complete set of classes upon which a class depends is the reflexive and transitive closure of the **directly depends on** relationship.

The example

```
class A: A {}
```

is erroneous because the class depends on itself. Likewise, the example

```
class A: B {}
class B: C {}
class C: A {}
```

is in error because the classes circularly depend on themselves. Finally, the example

```
class A: B.C {}

class B: A
{
    public class C {}
}
```

results in a compile-time error because `A` depends on `B.C` (its direct base class), which depends on `B` (its immediately enclosing class), which circularly depends on `A`.

Note that a class does not depend on the classes that are nested within it. In the example

```
class A
{
    class B: A {}
}
```

`B` depends on `A` (because `A` is both its direct base class and its immediately enclosing class), but `A` does not depend on `B` (since `B` is neither a base class nor an enclosing class of `A`). Thus, the example is valid.

It is not possible to derive from a `sealed` class. In the example

```
sealed class A {}

class B: A {}           // Error, cannot derive from a sealed class
```

class `B` is in error because it attempts to derive from the `sealed` class `A`.

Interface implementations

A *class_base* specification may include a list of interface types, in which case the class is said to directly implement the given interface types. Interface implementations are discussed further in [Interface implementations](#).

Type parameter constraints

Generic type and method declarations can optionally specify type parameter constraints by including *type_parameter_constraints_clauses*.


```

type_parameter_constraints_clause
: 'where' type_parameter ':' type_parameter_constraints
;

type_parameter_constraints
: primary_constraint
| secondary_constraints
| constructor_constraint
| primary_constraint ',' secondary_constraints
| primary_constraint ',' constructor_constraint
| secondary_constraints ',' constructor_constraint
| primary_constraint ',' secondary_constraints ',' constructor_constraint
;

primary_constraint
: class_type
| 'class'
| 'struct'
;

secondary_constraints
: interface_type
| type_parameter
| secondary_constraints ',' interface_type
| secondary_constraints ',' type_parameter
;

constructor_constraint
: 'new' '(' ')'
;

```

Each *type_parameter_constraints_clause* consists of the token `where`, followed by the name of a type parameter, followed by a colon and the list of constraints for that type parameter. There can be at most one `where` clause for each type parameter, and the `where` clauses can be listed in any order. Like the `get` and `set` tokens in a property accessor, the `where` token is not a keyword.

The list of constraints given in a `where` clause can include any of the following components, in this order: a single primary constraint, one or more secondary constraints, and the constructor constraint, `new()`.

A primary constraint can be a class type or the *reference type constraint* `class` or the *value type constraint* `struct`. A secondary constraint can be a *type_parameter* or *interface_type*.

The reference type constraint specifies that a type argument used for the type parameter must be a reference type. All class types, interface types, delegate types, array types, and type parameters known to be a reference type (as defined below) satisfy this constraint.

The value type constraint specifies that a type argument used for the type parameter must be a non-nullable value type. All non-nullable struct types, enum types, and type parameters having the value type constraint satisfy this constraint. Note that although classified as a value type, a nullable type ([Nullable types](#)) does not satisfy the value type constraint. A type parameter having the value type constraint cannot also have the *constructor_constraint*.

Pointer types are never allowed to be type arguments and are not considered to satisfy either the reference type or value type constraints.

If a constraint is a class type, an interface type, or a type parameter, that type specifies a minimal "base type" that every type argument used for that type parameter must support. Whenever a constructed type or generic method is used, the type argument is checked against the constraints on the type parameter at compile-time. The type argument supplied must satisfy the conditions described in [Satisfying constraints](#).

A *class_type* constraint must satisfy the following rules:

- The type must be a class type.
- The type must not be `sealed`.
- The type must not be one of the following types: `System.Array`, `System.Delegate`, `System.Enum`, or `System.ValueType`.
- The type must not be `object`. Because all types derive from `object`, such a constraint would have no effect if it were permitted.
- At most one constraint for a given type parameter can be a class type.

A type specified as an *interface_type* constraint must satisfy the following rules:

- The type must be an interface type.
- A type must not be specified more than once in a given `where` clause.

In either case, the constraint can involve any of the type parameters of the associated type or method declaration as part of a constructed type, and can involve the type being declared.

Any class or interface type specified as a type parameter constraint must be at least as accessible ([Accessibility constraints](#)) as the generic type or method being declared.

A type specified as a *type_parameter* constraint must satisfy the following rules:

- The type must be a type parameter.
- A type must not be specified more than once in a given `where` clause.

In addition there must be no cycles in the dependency graph of type parameters, where dependency is a transitive relation defined by:

- If a type parameter `T` is used as a constraint for type parameter `S` then `S` **depends on** `T`.
- If a type parameter `S` depends on a type parameter `T` and `T` depends on a type parameter `U` then `S` **depends on** `U`.

Given this relation, it is a compile-time error for a type parameter to depend on itself (directly or indirectly).

Any constraints must be consistent among dependent type parameters. If type parameter `S` depends on type parameter `T` then:

- `T` must not have the value type constraint. Otherwise, `T` is effectively sealed so `S` would be forced to be the same type as `T`, eliminating the need for two type parameters.
- If `S` has the value type constraint then `T` must not have a *class_type* constraint.
- If `S` has a *class_type* constraint `A` and `T` has a *class_type* constraint `B` then there must be an identity conversion or implicit reference conversion from `A` to `B` or an implicit reference conversion from `B` to `A`.
- If `S` also depends on type parameter `U` and `U` has a *class_type* constraint `A` and `T` has a *class_type* constraint `B` then there must be an identity conversion or implicit reference conversion from `A` to `B` or an implicit reference conversion from `B` to `A`.

It is valid for `S` to have the value type constraint and `T` to have the reference type constraint. Effectively this limits `T` to the types `System.Object`, `System.ValueType`, `System.Enum`, and any interface type.

If the `where` clause for a type parameter includes a constructor constraint (which has the form `new()`), it is possible to use the `new` operator to create instances of the type ([Object creation expressions](#)). Any type argument used for a type parameter with a constructor constraint must have a public parameterless constructor (this constructor implicitly exists for any value type) or be a type parameter having the value type constraint or constructor constraint (see [Type parameter constraints](#) for details).

The following are examples of constraints:

```

interface IPrintable
{
    void Print();
}

interface IComparable<T>
{
    int CompareTo(T value);
}

interface IKeyProvider<T>
{
    T GetKey();
}

class Printer<T> where T: IPrintable {...}

class SortedList<T> where T: IComparable<T> {...}

class Dictionary<K,V>
    where K: IComparable<K>
    where V: IPrintable, IKeyProvider<K>, new()
{
    ...
}

```

The following example is in error because it causes a circularity in the dependency graph of the type parameters:

```

class Circular<S,T>
    where S: T
    where T: S           // Error, circularity in dependency graph
{
    ...
}

```

The following examples illustrate additional invalid situations:

```

class Sealed<S,T>
    where S: T
    where T: struct       // Error, T is sealed
{
    ...
}

class A {...}

class B {...}

class Incompat<S,T>
    where S: A, T
    where T: B           // Error, incompatible class-type constraints
{
    ...
}

class StructWithClass<S,T,U>
    where S: struct, T
    where T: U
    where U: A           // Error, A incompatible with struct
{
    ...
}

```

The **effective base class** of a type parameter `T` is defined as follows:

- If `T` has no primary constraints or type parameter constraints, its effective base class is `object`.
- If `T` has the value type constraint, its effective base class is `System.ValueType`.
- If `T` has a *class_type* constraint `C` but no *type_parameter* constraints, its effective base class is `C`.
- If `T` has no *class_type* constraint but has one or more *type_parameter* constraints, its effective base class is the most encompassed type ([Lifted conversion operators](#)) in the set of effective base classes of its *type_parameter* constraints. The consistency rules ensure that such a most encompassed type exists.
- If `T` has both a *class_type* constraint and one or more *type_parameter* constraints, its effective base class is the most encompassed type ([Lifted conversion operators](#)) in the set consisting of the *class_type* constraint of `T` and the effective base classes of its *type_parameter* constraints. The consistency rules ensure that such a most encompassed type exists.
- If `T` has the reference type constraint but no *class_type* constraints, its effective base class is `object`.

For the purpose of these rules, if `T` has a constraint `V` that is a *value_type*, use instead the most specific base type of `V` that is a *class_type*. This can never happen in an explicitly given constraint, but may occur when the constraints of a generic method are implicitly inherited by an overriding method declaration or an explicit implementation of an interface method.

These rules ensure that the effective base class is always a *class_type*.

The **effective interface set** of a type parameter `T` is defined as follows:

- If `T` has no *secondary_constraints*, its effective interface set is empty.
- If `T` has *interface_type* constraints but no *type_parameter* constraints, its effective interface set is its set of *interface_type* constraints.
- If `T` has no *interface_type* constraints but has *type_parameter* constraints, its effective interface set is the union of the effective interface sets of its *type_parameter* constraints.
- If `T` has both *interface_type* constraints and *type_parameter* constraints, its effective interface set is the union of its set of *interface_type* constraints and the effective interface sets of its *type_parameter* constraints.

A type parameter is **known to be a reference type** if it has the reference type constraint or its effective base class is not `object` or `System.ValueType`.

Values of a constrained type parameter type can be used to access the instance members implied by the constraints. In the example

```
interface IPrintable
{
    void Print();
}

class Printer<T> where T: IPrintable
{
    void PrintOne(T x) {
        x.Print();
    }
}
```

the methods of `IPrintable` can be invoked directly on `x` because `T` is constrained to always implement `IPrintable`.

Class body

The *class_body* of a class defines the members of that class.

```
class_body
: '{' class_member_declaration* '}'
;
```

Partial types

A type declaration can be split across multiple *partial type declarations*. The type declaration is constructed from its parts by following the rules in this section, whereupon it is treated as a single declaration during the remainder of the compile-time and run-time processing of the program.

A *class_declaration*, *struct_declaration* or *interface_declaration* represents a partial type declaration if it includes a `partial` modifier. `partial` is not a keyword, and only acts as a modifier if it appears immediately before one of the keywords `class`, `struct` or `interface` in a type declaration, or before the type `void` in a method declaration. In other contexts it can be used as a normal identifier.

Each part of a partial type declaration must include a `partial` modifier. It must have the same name and be declared in the same namespace or type declaration as the other parts. The `partial` modifier indicates that additional parts of the type declaration may exist elsewhere, but the existence of such additional parts is not a requirement; it is valid for a type with a single declaration to include the `partial` modifier.

All parts of a partial type must be compiled together such that the parts can be merged at compile-time into a single type declaration. Partial types specifically do not allow already compiled types to be extended.

Nested types may be declared in multiple parts by using the `partial` modifier. Typically, the containing type is declared using `partial` as well, and each part of the nested type is declared in a different part of the containing type.

The `partial` modifier is not permitted on delegate or enum declarations.

Attributes

The attributes of a partial type are determined by combining, in an unspecified order, the attributes of each of the parts. If an attribute is placed on multiple parts, it is equivalent to specifying the attribute multiple times on the type. For example, the two parts:

```
[Attr1, Attr2("hello")]
partial class A {}

[Attr3, Attr2("goodbye")]
partial class A {}
```

are equivalent to a declaration such as:

```
[Attr1, Attr2("hello"), Attr3, Attr2("goodbye")]
class A {}
```

Attributes on type parameters combine in a similar fashion.

Modifiers

When a partial type declaration includes an accessibility specification (the `public`, `protected`, `internal`, and `private` modifiers) it must agree with all other parts that include an accessibility specification. If no part of a partial type includes an accessibility specification, the type is given the appropriate default accessibility ([Declared accessibility](#)).

If one or more partial declarations of a nested type include a `new` modifier, no warning is reported if the nested

type hides an inherited member ([Hiding through inheritance](#)).

If one or more partial declarations of a class include an `abstract` modifier, the class is considered abstract ([Abstract classes](#)). Otherwise, the class is considered non-abstract.

If one or more partial declarations of a class include a `sealed` modifier, the class is considered sealed ([Sealed classes](#)). Otherwise, the class is considered unsealed.

Note that a class cannot be both abstract and sealed.

When the `unsafe` modifier is used on a partial type declaration, only that particular part is considered an unsafe context ([Unsafe contexts](#)).

Type parameters and constraints

If a generic type is declared in multiple parts, each part must state the type parameters. Each part must have the same number of type parameters, and the same name for each type parameter, in order.

When a partial generic type declaration includes constraints (`where` clauses), the constraints must agree with all other parts that include constraints. Specifically, each part that includes constraints must have constraints for the same set of type parameters, and for each type parameter the sets of primary, secondary, and constructor constraints must be equivalent. Two sets of constraints are equivalent if they contain the same members. If no part of a partial generic type specifies type parameter constraints, the type parameters are considered unconstrained.

The example

```
partial class Dictionary<K,V>
    where K: IComparable<K>
    where V: IKeyProvider<K>, IPersistable
{
    ...
}

partial class Dictionary<K,V>
    where V: IPersistable, IKeyProvider<K>
    where K: IComparable<K>
{
    ...
}

partial class Dictionary<K,V>
{
    ...
}
```

is correct because those parts that include constraints (the first two) effectively specify the same set of primary, secondary, and constructor constraints for the same set of type parameters, respectively.

Base class

When a partial class declaration includes a base class specification it must agree with all other parts that include a base class specification. If no part of a partial class includes a base class specification, the base class becomes `System.Object` ([Base classes](#)).

Base interfaces

The set of base interfaces for a type declared in multiple parts is the union of the base interfaces specified on each part. A particular base interface may only be named once on each part, but it is permitted for multiple parts to name the same base interface(s). There must only be one implementation of the members of any given base interface.

In the example

```
partial class C: IA, IB {...}

partial class C: IC {...}

partial class C: IA, IB {...}
```

the set of base interfaces for class `C` is `IA`, `IB`, and `IC`.

Typically, each part provides an implementation of the interface(s) declared on that part; however, this is not a requirement. A part may provide the implementation for an interface declared on a different part:

```
partial class X
{
    int IComparable.CompareTo(object o) {...}
}

partial class X: IComparable
{
    ...
}
```

Members

With the exception of partial methods ([Partial methods](#)), the set of members of a type declared in multiple parts is simply the union of the set of members declared in each part. The bodies of all parts of the type declaration share the same declaration space ([Declarations](#)), and the scope of each member ([Scopes](#)) extends to the bodies of all the parts. The accessibility domain of any member always includes all the parts of the enclosing type; a `private` member declared in one part is freely accessible from another part. It is a compile-time error to declare the same member in more than one part of the type, unless that member is a type with the `partial` modifier.

```
partial class A
{
    int x;                // Error, cannot declare x more than once

    partial class Inner    // Ok, Inner is a partial type
    {
        int y;
    }
}

partial class A
{
    int x;                // Error, cannot declare x more than once

    partial class Inner    // Ok, Inner is a partial type
    {
        int z;
    }
}
```

The ordering of members within a type is rarely significant to C# code, but may be significant when interfacing with other languages and environments. In these cases, the ordering of members within a type declared in multiple parts is undefined.

Partial methods

Partial methods can be defined in one part of a type declaration and implemented in another. The

implementation is optional; if no part implements the partial method, the partial method declaration and all calls to it are removed from the type declaration resulting from the combination of the parts.

Partial methods cannot define access modifiers, but are implicitly `private`. Their return type must be `void`, and their parameters cannot have the `out` modifier. The identifier `partial` is recognized as a special keyword in a method declaration only if it appears right before the `void` type; otherwise it can be used as a normal identifier. A partial method cannot explicitly implement interface methods.

There are two kinds of partial method declarations: If the body of the method declaration is a semicolon, the declaration is said to be a **defining partial method declaration**. If the body is given as a *block*, the declaration is said to be an **implementing partial method declaration**. Across the parts of a type declaration there can be only one defining partial method declaration with a given signature, and there can be only one implementing partial method declaration with a given signature. If an implementing partial method declaration is given, a corresponding defining partial method declaration must exist, and the declarations must match as specified in the following:

- The declarations must have the same modifiers (although not necessarily in the same order), method name, number of type parameters and number of parameters.
- Corresponding parameters in the declarations must have the same modifiers (although not necessarily in the same order) and the same types (modulo differences in type parameter names).
- Corresponding type parameters in the declarations must have the same constraints (modulo differences in type parameter names).

An implementing partial method declaration can appear in the same part as the corresponding defining partial method declaration.

Only a defining partial method participates in overload resolution. Thus, whether or not an implementing declaration is given, invocation expressions may resolve to invocations of the partial method. Because a partial method always returns `void`, such invocation expressions will always be expression statements. Furthermore, because a partial method is implicitly `private`, such statements will always occur within one of the parts of the type declaration within which the partial method is declared.

If no part of a partial type declaration contains an implementing declaration for a given partial method, any expression statement invoking it is simply removed from the combined type declaration. Thus the invocation expression, including any constituent expressions, has no effect at run-time. The partial method itself is also removed and will not be a member of the combined type declaration.

If an implementing declaration exist for a given partial method, the invocations of the partial methods are retained. The partial method gives rise to a method declaration similar to the implementing partial method declaration except for the following:

- The `partial` modifier is not included
- The attributes in the resulting method declaration are the combined attributes of the defining and the implementing partial method declaration in unspecified order. Duplicates are not removed.
- The attributes on the parameters of the resulting method declaration are the combined attributes of the corresponding parameters of the defining and the implementing partial method declaration in unspecified order. Duplicates are not removed.

If a defining declaration but not an implementing declaration is given for a partial method M, the following restrictions apply:

- It is a compile-time error to create a delegate to method ([Delegate creation expressions](#)).
- It is a compile-time error to refer to `M` inside an anonymous function that is converted to an expression tree type ([Evaluation of anonymous function conversions to expression tree types](#)).
- Expressions occurring as part of an invocation of `M` do not affect the definite assignment state ([Definite](#)

[assignment](#)), which can potentially lead to compile-time errors.

- `M` cannot be the entry point for an application ([Application Startup](#)).

Partial methods are useful for allowing one part of a type declaration to customize the behavior of another part, e.g., one that is generated by a tool. Consider the following partial class declaration:

```
partial class Customer
{
    string name;

    public string Name {
        get { return name; }
        set {
            OnNameChanging(value);
            name = value;
            OnNameChanged();
        }
    }

    partial void OnNameChanging(string newName);

    partial void OnNameChanged();
}
```

If this class is compiled without any other parts, the defining partial method declarations and their invocations will be removed, and the resulting combined class declaration will be equivalent to the following:

```
class Customer
{
    string name;

    public string Name {
        get { return name; }
        set { name = value; }
    }
}
```

Assume that another part is given, however, which provides implementing declarations of the partial methods:

```
partial class Customer
{
    partial void OnNameChanging(string newName)
    {
        Console.WriteLine("Changing " + name + " to " + newName);
    }

    partial void OnNameChanged()
    {
        Console.WriteLine("Changed to " + name);
    }
}
```

Then the resulting combined class declaration will be equivalent to the following:

```

class Customer
{
    string name;

    public string Name {
        get { return name; }
        set {
            OnNameChanging(value);
            name = value;
            OnNameChanged();
        }
    }

    void OnNameChanging(string newName)
    {
        Console.WriteLine("Changing " + name + " to " + newName);
    }

    void OnNameChanged()
    {
        Console.WriteLine("Changed to " + name);
    }
}

```

Name binding

Although each part of an extensible type must be declared within the same namespace, the parts are typically written within different namespace declarations. Thus, different `using` directives ([Using directives](#)) may be present for each part. When interpreting simple names ([Type inference](#)) within one part, only the `using` directives of the namespace declaration(s) enclosing that part are considered. This may result in the same identifier having different meanings in different parts:

```

namespace N
{
    using List = System.Collections.ArrayList;

    partial class A
    {
        List x;           // x has type System.Collections.ArrayList
    }
}

namespace N
{
    using List = Widgets.LinkedList;

    partial class A
    {
        List y;           // y has type Widgets.LinkedList
    }
}

```

Class members

The members of a class consist of the members introduced by its *class_member_declarations* and the members inherited from the direct base class.

```

class_member_declaration
: constant_declaration
| field_declaration
| method_declaration
| property_declaration
| event_declaration
| indexer_declaration
| operator_declaration
| constructor_declaration
| destructor_declaration
| static_constructor_declaration
| type_declaration
;

```

The members of a class type are divided into the following categories:

- Constants, which represent constant values associated with the class ([Constants](#)).
- Fields, which are the variables of the class ([Fields](#)).
- Methods, which implement the computations and actions that can be performed by the class ([Methods](#)).
- Properties, which define named characteristics and the actions associated with reading and writing those characteristics ([Properties](#)).
- Events, which define notifications that can be generated by the class ([Events](#)).
- Indexers, which permit instances of the class to be indexed in the same way (syntactically) as arrays ([Indexers](#)).
- Operators, which define the expression operators that can be applied to instances of the class ([Operators](#)).
- Instance constructors, which implement the actions required to initialize instances of the class ([Instance constructors](#)).
- Destructors, which implement the actions to be performed before instances of the class are permanently discarded ([Destructors](#)).
- Static constructors, which implement the actions required to initialize the class itself ([Static constructors](#)).
- Types, which represent the types that are local to the class ([Nested types](#)).

Members that can contain executable code are collectively known as the *function members* of the class type. The function members of a class type are the methods, properties, events, indexers, operators, instance constructors, destructors, and static constructors of that class type.

A *class_declaration* creates a new declaration space ([Declarations](#)), and the *class_member_declarations* immediately contained by the *class_declaration* introduce new members into this declaration space. The following rules apply to *class_member_declarations*:

- Instance constructors, destructors and static constructors must have the same name as the immediately enclosing class. All other members must have names that differ from the name of the immediately enclosing class.
- The name of a constant, field, property, event, or type must differ from the names of all other members declared in the same class.
- The name of a method must differ from the names of all other non-methods declared in the same class. In addition, the signature ([Signatures and overloading](#)) of a method must differ from the signatures of all other methods declared in the same class, and two methods declared in the same class may not have signatures that differ solely by `ref` and `out`.
- The signature of an instance constructor must differ from the signatures of all other instance constructors declared in the same class, and two constructors declared in the same class may not have signatures that differ solely by `ref` and `out`.
- The signature of an indexer must differ from the signatures of all other indexers declared in the same class.
- The signature of an operator must differ from the signatures of all other operators declared in the same

class.

The inherited members of a class type ([Inheritance](#)) are not part of the declaration space of a class. Thus, a derived class is allowed to declare a member with the same name or signature as an inherited member (which in effect hides the inherited member).

The instance type

Each class declaration has an associated bound type ([Bound and unbound types](#)), the *instance type*. For a generic class declaration, the instance type is formed by creating a constructed type ([Constructed types](#)) from the type declaration, with each of the supplied type arguments being the corresponding type parameter. Since the instance type uses the type parameters, it can only be used where the type parameters are in scope; that is, inside the class declaration. The instance type is the type of `this` for code written inside the class declaration. For non-generic classes, the instance type is simply the declared class. The following shows several class declarations along with their instance types:

```
class A<T>                // instance type: A<T>
{
    class B {}            // instance type: A<T>.B
    class C<U> {}         // instance type: A<T>.C<U>
}

class D {}                // instance type: D
```

Members of constructed types

The non-inherited members of a constructed type are obtained by substituting, for each *type_parameter* in the member declaration, the corresponding *type_argument* of the constructed type. The substitution process is based on the semantic meaning of type declarations, and is not simply textual substitution.

For example, given the generic class declaration

```
class Gen<T,U>
{
    public T[, ] a;
    public void G(int i, T t, Gen<U,T> gt) {...}
    public U Prop { get {...} set {...} }
    public int H(double d) {...}
}
```

the constructed type `Gen<int[],IComparable<string>>` has the following members:

```
public int[, ][ ] a;
public void G(int i, int[] t, Gen<IComparable<string>,int[]> gt) {...}
public IComparable<string> Prop { get {...} set {...} }
public int H(double d) {...}
```

The type of the member `a` in the generic class declaration `Gen` is "two-dimensional array of `T`", so the type of the member `a` in the constructed type above is "two-dimensional array of one-dimensional array of `int`", or `int[,][]`.

Within instance function members, the type of `this` is the instance type ([The instance type](#)) of the containing declaration.

All members of a generic class can use type parameters from any enclosing class, either directly or as part of a constructed type. When a particular closed constructed type ([Open and closed types](#)) is used at run-time, each use of a type parameter is replaced with the actual type argument supplied to the constructed type. For example:

```

class C<V>
{
    public V f1;
    public C<V> f2 = null;

    public C(V x) {
        this.f1 = x;
        this.f2 = this;
    }
}

class Application
{
    static void Main() {
        C<int> x1 = new C<int>(1);
        Console.WriteLine(x1.f1);          // Prints 1

        C<double> x2 = new C<double>(3.1415);
        Console.WriteLine(x2.f1);          // Prints 3.1415
    }
}

```

Inheritance

A class *inherits* the members of its direct base class type. Inheritance means that a class implicitly contains all members of its direct base class type, except for the instance constructors, destructors and static constructors of the base class. Some important aspects of inheritance are:

- Inheritance is transitive. If `C` is derived from `B`, and `B` is derived from `A`, then `C` inherits the members declared in `B` as well as the members declared in `A`.
- A derived class extends its direct base class. A derived class can add new members to those it inherits, but it cannot remove the definition of an inherited member.
- Instance constructors, destructors, and static constructors are not inherited, but all other members are, regardless of their declared accessibility ([Member access](#)). However, depending on their declared accessibility, inherited members might not be accessible in a derived class.
- A derived class can *hide* ([Hiding through inheritance](#)) inherited members by declaring new members with the same name or signature. Note however that hiding an inherited member does not remove that member—it merely makes that member inaccessible directly through the derived class.
- An instance of a class contains a set of all instance fields declared in the class and its base classes, and an implicit conversion ([Implicit reference conversions](#)) exists from a derived class type to any of its base class types. Thus, a reference to an instance of some derived class can be treated as a reference to an instance of any of its base classes.
- A class can declare virtual methods, properties, and indexers, and derived classes can override the implementation of these function members. This enables classes to exhibit polymorphic behavior wherein the actions performed by a function member invocation varies depending on the run-time type of the instance through which that function member is invoked.

The inherited member of a constructed class type are the members of the immediate base class type ([Base classes](#)), which is found by substituting the type arguments of the constructed type for each occurrence of the corresponding type parameters in the *class_base* specification. These members, in turn, are transformed by substituting, for each *type_parameter* in the member declaration, the corresponding *type_argument* of the *class_base* specification.

```

class B<U>
{
    public U F(long index) {...}
}

class D<T>: B<T[]>
{
    public T G(string s) {...}
}

```

In the above example, the constructed type `D<int>` has a non-inherited member `public int G(string s)` obtained by substituting the type argument `int` for the type parameter `T`. `D<int>` also has an inherited member from the class declaration `B`. This inherited member is determined by first determining the base class type `B<int[]>` of `D<int>` by substituting `int` for `T` in the base class specification `B<T[]>`. Then, as a type argument to `B`, `int[]` is substituted for `U` in `public U F(long index)`, yielding the inherited member `public int[] F(long index)`.

The new modifier

A *class_member_declaration* is permitted to declare a member with the same name or signature as an inherited member. When this occurs, the derived class member is said to **hide** the base class member. Hiding an inherited member is not considered an error, but it does cause the compiler to issue a warning. To suppress the warning, the declaration of the derived class member can include a `new` modifier to indicate that the derived member is intended to hide the base member. This topic is discussed further in [Hiding through inheritance](#).

If a `new` modifier is included in a declaration that doesn't hide an inherited member, a warning to that effect is issued. This warning is suppressed by removing the `new` modifier.

Access modifiers

A *class_member_declaration* can have any one of the five possible kinds of declared accessibility ([Declared accessibility](#)): `public`, `protected internal`, `protected`, `internal`, or `private`. Except for the `protected internal` combination, it is a compile-time error to specify more than one access modifier. When a *class_member_declaration* does not include any access modifiers, `private` is assumed.

Constituent types

Types that are used in the declaration of a member are called the constituent types of that member. Possible constituent types are the type of a constant, field, property, event, or indexer, the return type of a method or operator, and the parameter types of a method, indexer, operator, or instance constructor. The constituent types of a member must be at least as accessible as that member itself ([Accessibility constraints](#)).

Static and instance members

Members of a class are either **static members** or **instance members**. Generally speaking, it is useful to think of static members as belonging to class types and instance members as belonging to objects (instances of class types).

When a field, method, property, event, operator, or constructor declaration includes a `static` modifier, it declares a static member. In addition, a constant or type declaration implicitly declares a static member. Static members have the following characteristics:

- When a static member `M` is referenced in a *member_access* ([Member access](#)) of the form `E.M`, `E` must denote a type containing `M`. It is a compile-time error for `E` to denote an instance.
- A static field identifies exactly one storage location to be shared by all instances of a given closed class type. No matter how many instances of a given closed class type are created, there is only ever one copy of a static field.
- A static function member (method, property, event, operator, or constructor) does not operate on a specific

instance, and it is a compile-time error to refer to `this` in such a function member.

When a field, method, property, event, indexer, constructor, or destructor declaration does not include a `static` modifier, it declares an instance member. (An instance member is sometimes called a non-static member.) Instance members have the following characteristics:

- When an instance member `M` is referenced in a *member_access* (Member access) of the form `E.M`, `E` must denote an instance of a type containing `M`. It is a binding-time error for `E` to denote a type.
- Every instance of a class contains a separate set of all instance fields of the class.
- An instance function member (method, property, indexer, instance constructor, or destructor) operates on a given instance of the class, and this instance can be accessed as `this` (This access).

The following example illustrates the rules for accessing static and instance members:

```
class Test
{
    int x;
    static int y;

    void F() {
        x = 1;           // Ok, same as this.x = 1
        y = 1;           // Ok, same as Test.y = 1
    }

    static void G() {
        x = 1;           // Error, cannot access this.x
        y = 1;           // Ok, same as Test.y = 1
    }

    static void Main() {
        Test t = new Test();
        t.x = 1;         // Ok
        t.y = 1;         // Error, cannot access static member through instance
        Test.x = 1;      // Error, cannot access instance member through type
        Test.y = 1;      // Ok
    }
}
```

The `F` method shows that in an instance function member, a *simple_name* (Simple names) can be used to access both instance members and static members. The `G` method shows that in a static function member, it is a compile-time error to access an instance member through a *simple_name*. The `Main` method shows that in a *member_access* (Member access), instance members must be accessed through instances, and static members must be accessed through types.

Nested types

A type declared within a class or struct declaration is called a *nested type*. A type that is declared within a compilation unit or namespace is called a *non-nested type*.

In the example

```

using System;

class A
{
    class B
    {
        static void F() {
            Console.WriteLine("A.B.F");
        }
    }
}

```

class `B` is a nested type because it is declared within class `A`, and class `A` is a non-nested type because it is declared within a compilation unit.

Fully qualified name

The fully qualified name ([Fully qualified names](#)) for a nested type is `S.N` where `S` is the fully qualified name of the type in which type `N` is declared.

Declared accessibility

Non-nested types can have `public` or `internal` declared accessibility and have `internal` declared accessibility by default. Nested types can have these forms of declared accessibility too, plus one or more additional forms of declared accessibility, depending on whether the containing type is a class or struct:

- A nested type that is declared in a class can have any of five forms of declared accessibility (`public`, `protected internal`, `protected`, `internal`, or `private`) and, like other class members, defaults to `private` declared accessibility.
- A nested type that is declared in a struct can have any of three forms of declared accessibility (`public`, `internal`, or `private`) and, like other struct members, defaults to `private` declared accessibility.

The example

```

public class List
{
    // Private data structure
    private class Node
    {
        public object Data;
        public Node Next;

        public Node(object data, Node next) {
            this.Data = data;
            this.Next = next;
        }
    }

    private Node first = null;
    private Node last = null;

    // Public interface
    public void AddToFront(object o) {...}
    public void AddToBack(object o) {...}
    public object RemoveFromFront() {...}
    public object RemoveFromBack() {...}
    public int Count { get {...} }
}

```

declares a private nested class `Node`.

Hiding

A nested type may hide ([Name hiding](#)) a base member. The `new` modifier is permitted on nested type declarations so that hiding can be expressed explicitly. The example

```
using System;

class Base
{
    public static void M() {
        Console.WriteLine("Base.M");
    }
}

class Derived: Base
{
    new public class M
    {
        public static void F() {
            Console.WriteLine("Derived.M.F");
        }
    }
}

class Test
{
    static void Main() {
        Derived.M.F();
    }
}
```

shows a nested class `M` that hides the method `M` defined in `Base`.

this access

A nested type and its containing type do not have a special relationship with regard to *this_access* ([This access](#)). Specifically, `this` within a nested type cannot be used to refer to instance members of the containing type. In cases where a nested type needs access to the instance members of its containing type, access can be provided by providing the `this` for the instance of the containing type as a constructor argument for the nested type. The following example

```

using System;

class C
{
    int i = 123;

    public void F() {
        Nested n = new Nested(this);
        n.G();
    }

    public class Nested
    {
        C this_c;

        public Nested(C c) {
            this_c = c;
        }

        public void G() {
            Console.WriteLine(this_c.i);
        }
    }
}

class Test
{
    static void Main() {
        C c = new C();
        c.F();
    }
}

```

shows this technique. An instance of `C` creates an instance of `Nested` and passes its own `this` to `Nested`'s constructor in order to provide subsequent access to `C`'s instance members.

Access to private and protected members of the containing type

A nested type has access to all of the members that are accessible to its containing type, including members of the containing type that have `private` and `protected` declared accessibility. The example

```

using System;

class C
{
    private static void F() {
        Console.WriteLine("C.F");
    }

    public class Nested
    {
        public static void G() {
            F();
        }
    }
}

class Test
{
    static void Main() {
        C.Nested.G();
    }
}

```

shows a class `C` that contains a nested class `Nested`. Within `Nested`, the method `G` calls the static method `F` defined in `C`, and `F` has private declared accessibility.

A nested type also may access protected members defined in a base type of its containing type. In the example

```
using System;

class Base
{
    protected void F() {
        Console.WriteLine("Base.F");
    }
}

class Derived: Base
{
    public class Nested
    {
        public void G() {
            Derived d = new Derived();
            d.F();        // ok
        }
    }
}

class Test
{
    static void Main() {
        Derived.Nested n = new Derived.Nested();
        n.G();
    }
}
```

the nested class `Derived.Nested` accesses the protected method `F` defined in `Derived`'s base class, `Base`, by calling through an instance of `Derived`.

Nested types in generic classes

A generic class declaration can contain nested type declarations. The type parameters of the enclosing class can be used within the nested types. A nested type declaration can contain additional type parameters that apply only to the nested type.

Every type declaration contained within a generic class declaration is implicitly a generic type declaration. When writing a reference to a type nested within a generic type, the containing constructed type, including its type arguments, must be named. However, from within the outer class, the nested type can be used without qualification; the instance type of the outer class can be implicitly used when constructing the nested type. The following example shows three different correct ways to refer to a constructed type created from `Inner`; the first two are equivalent:

```

class Outer<T>
{
    class Inner<U>
    {
        public static void F(T t, U u) {...}
    }

    static void F(T t) {
        Outer<T>.Inner<string>.F(t, "abc");    // These two statements have
        Inner<string>.F(t, "abc");              // the same effect

        Outer<int>.Inner<string>.F(3, "abc");    // This type is different

        Outer.Inner<string>.F(t, "abc");        // Error, Outer needs type arg
    }
}

```

Although it is bad programming style, a type parameter in a nested type can hide a member or type parameter declared in the outer type:

```

class Outer<T>
{
    class Inner<T>          // Valid, hides Outer's T
    {
        public T t;        // Refers to Inner's T
    }
}

```

Reserved member names

To facilitate the underlying C# run-time implementation, for each source member declaration that is a property, event, or indexer, the implementation must reserve two method signatures based on the kind of the member declaration, its name, and its type. It is a compile-time error for a program to declare a member whose signature matches one of these reserved signatures, even if the underlying run-time implementation does not make use of these reservations.

The reserved names do not introduce declarations, thus they do not participate in member lookup. However, a declaration's associated reserved method signatures do participate in inheritance ([Inheritance](#)), and can be hidden with the `new` modifier ([The new modifier](#)).

The reservation of these names serves three purposes:

- To allow the underlying implementation to use an ordinary identifier as a method name for get or set access to the C# language feature.
- To allow other languages to interoperate using an ordinary identifier as a method name for get or set access to the C# language feature.
- To help ensure that the source accepted by one conforming compiler is accepted by another, by making the specifics of reserved member names consistent across all C# implementations.

The declaration of a destructor ([Destructors](#)) also causes a signature to be reserved ([Member names reserved for destructors](#)).

Member names reserved for properties

For a property `P` ([Properties](#)) of type `T`, the following signatures are reserved:

```

T get_P();
void set_P(T value);

```

Both signatures are reserved, even if the property is read-only or write-only.

In the example

```
using System;

class A
{
    public int P {
        get { return 123; }
    }
}

class B: A
{
    new public int get_P() {
        return 456;
    }

    new public void set_P(int value) {
    }
}

class Test
{
    static void Main() {
        B b = new B();
        A a = b;
        Console.WriteLine(a.P);
        Console.WriteLine(b.P);
        Console.WriteLine(b.get_P());
    }
}
```

a class `A` defines a read-only property `P`, thus reserving signatures for `get_P` and `set_P` methods. A class `B` derives from `A` and hides both of these reserved signatures. The example produces the output:

```
123
123
456
```

Member names reserved for events

For an event `E` ([Events](#)) of delegate type `T`, the following signatures are reserved:

```
void add_E(T handler);
void remove_E(T handler);
```

Member names reserved for indexers

For an indexer ([Indexers](#)) of type `T` with parameter-list `L`, the following signatures are reserved:

```
T get_Item(L);
void set_Item(L, T value);
```

Both signatures are reserved, even if the indexer is read-only or write-only.

Furthermore the member name `Item` is reserved.

Member names reserved for destructors

For a class containing a destructor ([Destructors](#)), the following signature is reserved:

```
void Finalize();
```

Constants

A **constant** is a class member that represents a constant value: a value that can be computed at compile-time. A *constant_declaration* introduces one or more constants of a given type.

```
constant_declaration
    : attributes? constant_modifier* 'const' type constant_declarators ';'
    ;

constant_modifier
    : 'new'
    | 'public'
    | 'protected'
    | 'internal'
    | 'private'
    ;

constant_declarators
    : constant_declarator (',' constant_declarator)*
    ;

constant_declarator
    : identifier '=' constant_expression
    ;
```

A *constant_declaration* may include a set of *attributes* ([Attributes](#)), a `new` modifier ([The new modifier](#)), and a valid combination of the four access modifiers ([Access modifiers](#)). The attributes and modifiers apply to all of the members declared by the *constant_declaration*. Even though constants are considered static members, a *constant_declaration* neither requires nor allows a `static` modifier. It is an error for the same modifier to appear multiple times in a constant declaration.

The *type* of a *constant_declaration* specifies the type of the members introduced by the declaration. The type is followed by a list of *constant_declarators*, each of which introduces a new member. A *constant_declarator* consists of an *identifier* that names the member, followed by an "=" token, followed by a *constant_expression* ([Constant expressions](#)) that gives the value of the member.

The *type* specified in a constant declaration must be `sbyte`, `byte`, `short`, `ushort`, `int`, `uint`, `long`, `ulong`, `char`, `float`, `double`, `decimal`, `bool`, `string`, an *enum_type*, or a *reference_type*. Each *constant_expression* must yield a value of the target type or of a type that can be converted to the target type by an implicit conversion ([Implicit conversions](#)).

The *type* of a constant must be at least as accessible as the constant itself ([Accessibility constraints](#)).

The value of a constant is obtained in an expression using a *simple_name* ([Simple names](#)) or a *member_access* ([Member access](#)).

A constant can itself participate in a *constant_expression*. Thus, a constant may be used in any construct that requires a *constant_expression*. Examples of such constructs include `case` labels, `goto case` statements, `enum` member declarations, attributes, and other constant declarations.

As described in [Constant expressions](#), a *constant_expression* is an expression that can be fully evaluated at compile-time. Since the only way to create a non-null value of a *reference_type* other than `string` is to apply the `new` operator, and since the `new` operator is not permitted in a *constant_expression*, the only possible value for constants of *reference_types* other than `string` is `null`.

When a symbolic name for a constant value is desired, but when the type of that value is not permitted in a constant declaration, or when the value cannot be computed at compile-time by a *constant_expression*, a `readonly` field ([Readonly fields](#)) may be used instead.

A constant declaration that declares multiple constants is equivalent to multiple declarations of single constants with the same attributes, modifiers, and type. For example

```
class A
{
    public const double X = 1.0, Y = 2.0, Z = 3.0;
}
```

is equivalent to

```
class A
{
    public const double X = 1.0;
    public const double Y = 2.0;
    public const double Z = 3.0;
}
```

Constants are permitted to depend on other constants within the same program as long as the dependencies are not of a circular nature. The compiler automatically arranges to evaluate the constant declarations in the appropriate order. In the example

```
class A
{
    public const int X = B.Z + 1;
    public const int Y = 10;
}

class B
{
    public const int Z = A.Y + 1;
}
```

the compiler first evaluates `A.Y`, then evaluates `B.Z`, and finally evaluates `A.X`, producing the values `10`, `11`, and `12`. Constant declarations may depend on constants from other programs, but such dependencies are only possible in one direction. Referring to the example above, if `A` and `B` were declared in separate programs, it would be possible for `A.X` to depend on `B.Z`, but `B.Z` could then not simultaneously depend on `A.Y`.

Fields

A *field* is a member that represents a variable associated with an object or class. A *field_declaration* introduces one or more fields of a given type.

```

field_declaration
: attributes? field_modifier* type variable_declarators ';'
;

field_modifier
: 'new'
| 'public'
| 'protected'
| 'internal'
| 'private'
| 'static'
| 'readonly'
| 'volatile'
| field_modifier_unsafe
;

variable_declarators
: variable_declarator (',' variable_declarator)*
;

variable_declarator
: identifier ('=' variable_initializer)?
;

variable_initializer
: expression
| array_initializer
;

```

A *field_declaration* may include a set of *attributes* ([Attributes](#)), a `new` modifier ([The new modifier](#)), a valid combination of the four access modifiers ([Access modifiers](#)), and a `static` modifier ([Static and instance fields](#)). In addition, a *field_declaration* may include a `readonly` modifier ([Readonly fields](#)) or a `volatile` modifier ([Volatile fields](#)) but not both. The attributes and modifiers apply to all of the members declared by the *field_declaration*. It is an error for the same modifier to appear multiple times in a field declaration.

The *type* of a *field_declaration* specifies the type of the members introduced by the declaration. The type is followed by a list of *variable_declarators*, each of which introduces a new member. A *variable_declarator* consists of an *identifier* that names that member, optionally followed by an "=" token and a *variable_initializer* ([Variable initializers](#)) that gives the initial value of that member.

The *type* of a field must be at least as accessible as the field itself ([Accessibility constraints](#)).

The value of a field is obtained in an expression using a *simple_name* ([Simple names](#)) or a *member_access* ([Member access](#)). The value of a non-readonly field is modified using an *assignment* ([Assignment operators](#)). The value of a non-readonly field can be both obtained and modified using postfix increment and decrement operators ([Postfix increment and decrement operators](#)) and prefix increment and decrement operators ([Prefix increment and decrement operators](#)).

A field declaration that declares multiple fields is equivalent to multiple declarations of single fields with the same attributes, modifiers, and type. For example

```

class A
{
    public static int X = 1, Y, Z = 100;
}

```

is equivalent to


```

class A
{
    public static int X = 1;
    public static int Y;
    public static int Z = 100;
}

```

Static and instance fields

When a field declaration includes a `static` modifier, the fields introduced by the declaration are *static fields*. When no `static` modifier is present, the fields introduced by the declaration are *instance fields*. Static fields and instance fields are two of the several kinds of variables ([Variables](#)) supported by C#, and at times they are referred to as *static variables* and *instance variables*, respectively.

A static field is not part of a specific instance; instead, it is shared amongst all instances of a closed type ([Open and closed types](#)). No matter how many instances of a closed class type are created, there is only ever one copy of a static field for the associated application domain.

For example:

```

class C<V>
{
    static int count = 0;

    public C() {
        count++;
    }

    public static int Count {
        get { return count; }
    }
}

class Application
{
    static void Main() {
        C<int> x1 = new C<int>();
        Console.WriteLine(C<int>.Count);           // Prints 1

        C<double> x2 = new C<double>();
        Console.WriteLine(C<int>.Count);           // Prints 1

        C<int> x3 = new C<int>();
        Console.WriteLine(C<int>.Count);           // Prints 2
    }
}

```

An instance field belongs to an instance. Specifically, every instance of a class contains a separate set of all the instance fields of that class.

When a field is referenced in a *member access* ([Member access](#)) of the form `E.M`, if `M` is a static field, `E` must denote a type containing `M`, and if `M` is an instance field, `E` must denote an instance of a type containing `M`.

The differences between static and instance members are discussed further in [Static and instance members](#).

ReadOnly fields

When a *field declaration* includes a `readonly` modifier, the fields introduced by the declaration are *readonly fields*. Direct assignments to readonly fields can only occur as part of that declaration or in an instance constructor or static constructor in the same class. (A readonly field can be assigned to multiple times in these contexts.) Specifically, direct assignments to a `readonly` field are permitted only in the following contexts:

- In the *variable_declarator* that introduces the field (by including a *variable_initializer* in the declaration).
- For an instance field, in the instance constructors of the class that contains the field declaration; for a static field, in the static constructor of the class that contains the field declaration. These are also the only contexts in which it is valid to pass a `readonly` field as an `out` or `ref` parameter.

Attempting to assign to a `readonly` field or pass it as an `out` or `ref` parameter in any other context is a compile-time error.

Using static readonly fields for constants

A `static readonly` field is useful when a symbolic name for a constant value is desired, but when the type of the value is not permitted in a `const` declaration, or when the value cannot be computed at compile-time. In the example

```
public class Color
{
    public static readonly Color Black = new Color(0, 0, 0);
    public static readonly Color White = new Color(255, 255, 255);
    public static readonly Color Red = new Color(255, 0, 0);
    public static readonly Color Green = new Color(0, 255, 0);
    public static readonly Color Blue = new Color(0, 0, 255);

    private byte red, green, blue;

    public Color(byte r, byte g, byte b) {
        red = r;
        green = g;
        blue = b;
    }
}
```

the `Black`, `White`, `Red`, `Green`, and `Blue` members cannot be declared as `const` members because their values cannot be computed at compile-time. However, declaring them `static readonly` instead has much the same effect.

Versioning of constants and static readonly fields

Constants and readonly fields have different binary versioning semantics. When an expression references a constant, the value of the constant is obtained at compile-time, but when an expression references a readonly field, the value of the field is not obtained until run-time. Consider an application that consists of two separate programs:

```
using System;

namespace Program1
{
    public class Utils
    {
        public static readonly int X = 1;
    }
}

namespace Program2
{
    class Test
    {
        static void Main() {
            Console.WriteLine(Program1.Utils.X);
        }
    }
}
```

The `Program1` and `Program2` namespaces denote two programs that are compiled separately. Because `Program1.Utils.X` is declared as a static readonly field, the value output by the `Console.WriteLine` statement is not known at compile-time, but rather is obtained at run-time. Thus, if the value of `x` is changed and `Program1` is recompiled, the `Console.WriteLine` statement will output the new value even if `Program2` isn't recompiled. However, had `x` been a constant, the value of `x` would have been obtained at the time `Program2` was compiled, and would remain unaffected by changes in `Program1` until `Program2` is recompiled.

Volatile fields

When a *field_declaration* includes a `volatile` modifier, the fields introduced by that declaration are *volatile fields*.

For non-volatile fields, optimization techniques that reorder instructions can lead to unexpected and unpredictable results in multi-threaded programs that access fields without synchronization such as that provided by the *lock_statement* ([The lock statement](#)). These optimizations can be performed by the compiler, by the run-time system, or by hardware. For volatile fields, such reordering optimizations are restricted:

- A read of a volatile field is called a *volatile read*. A volatile read has "acquire semantics"; that is, it is guaranteed to occur prior to any references to memory that occur after it in the instruction sequence.
- A write of a volatile field is called a *volatile write*. A volatile write has "release semantics"; that is, it is guaranteed to happen after any memory references prior to the write instruction in the instruction sequence.

These restrictions ensure that all threads will observe volatile writes performed by any other thread in the order in which they were performed. A conforming implementation is not required to provide a single total ordering of volatile writes as seen from all threads of execution. The type of a volatile field must be one of the following:

- A *reference_type*.
- The type `byte`, `sbyte`, `short`, `ushort`, `int`, `uint`, `char`, `float`, `bool`, `System.IntPtr`, or `System.UIntPtr`.
- An *enum_type* having an enum base type of `byte`, `sbyte`, `short`, `ushort`, `int`, or `uint`.

The example

```

using System;
using System.Threading;

class Test
{
    public static int result;
    public static volatile bool finished;

    static void Thread2() {
        result = 143;
        finished = true;
    }

    static void Main() {
        finished = false;

        // Run Thread2() in a new thread
        new Thread(new ThreadStart(Thread2)).Start();

        // Wait for Thread2 to signal that it has a result by setting
        // finished to true.
        for (;;) {
            if (finished) {
                Console.WriteLine("result = {0}", result);
                return;
            }
        }
    }
}

```

produces the output:

```

result = 143

```

In this example, the method `Main` starts a new thread that runs the method `Thread2`. This method stores a value into a non-volatile field called `result`, then stores `true` in the volatile field `finished`. The main thread waits for the field `finished` to be set to `true`, then reads the field `result`. Since `finished` has been declared `volatile`, the main thread must read the value `143` from the field `result`. If the field `finished` had not been declared `volatile`, then it would be permissible for the store to `result` to be visible to the main thread after the store to `finished`, and hence for the main thread to read the value `0` from the field `result`. Declaring `finished` as a `volatile` field prevents any such inconsistency.

Field initialization

The initial value of a field, whether it be a static field or an instance field, is the default value ([Default values](#)) of the field's type. It is not possible to observe the value of a field before this default initialization has occurred, and a field is thus never "uninitialized". The example

```

using System;

class Test
{
    static bool b;
    int i;

    static void Main() {
        Test t = new Test();
        Console.WriteLine("b = {0}, i = {1}", b, t.i);
    }
}

```

produces the output

```
b = False, i = 0
```

because `b` and `i` are both automatically initialized to default values.

Variable initializers

Field declarations may include *variable initializers*. For static fields, variable initializers correspond to assignment statements that are executed during class initialization. For instance fields, variable initializers correspond to assignment statements that are executed when an instance of the class is created.

The example

```
using System;

class Test
{
    static double x = Math.Sqrt(2.0);
    int i = 100;
    string s = "Hello";

    static void Main() {
        Test a = new Test();
        Console.WriteLine("x = {0}, i = {1}, s = {2}", x, a.i, a.s);
    }
}
```

produces the output

```
x = 1.4142135623731, i = 100, s = Hello
```

because an assignment to `x` occurs when static field initializers execute and assignments to `i` and `s` occur when the instance field initializers execute.

The default value initialization described in [Field initialization](#) occurs for all fields, including fields that have variable initializers. Thus, when a class is initialized, all static fields in that class are first initialized to their default values, and then the static field initializers are executed in textual order. Likewise, when an instance of a class is created, all instance fields in that instance are first initialized to their default values, and then the instance field initializers are executed in textual order.

It is possible for static fields with variable initializers to be observed in their default value state. However, this is strongly discouraged as a matter of style. The example

```
using System;

class Test
{
    static int a = b + 1;
    static int b = a + 1;

    static void Main() {
        Console.WriteLine("a = {0}, b = {1}", a, b);
    }
}
```

exhibits this behavior. Despite the circular definitions of `a` and `b`, the program is valid. It results in the output

```
a = 1, b = 2
```

because the static fields `a` and `b` are initialized to `0` (the default value for `int`) before their initializers are executed. When the initializer for `a` runs, the value of `b` is zero, and so `a` is initialized to `1`. When the initializer for `b` runs, the value of `a` is already `1`, and so `b` is initialized to `2`.

Static field initialization

The static field variable initializers of a class correspond to a sequence of assignments that are executed in the textual order in which they appear in the class declaration. If a static constructor ([Static constructors](#)) exists in the class, execution of the static field initializers occurs immediately prior to executing that static constructor. Otherwise, the static field initializers are executed at an implementation-dependent time prior to the first use of a static field of that class. The example

```
using System;

class Test
{
    static void Main() {
        Console.WriteLine("{0} {1}", B.Y, A.X);
    }

    public static int F(string s) {
        Console.WriteLine(s);
        return 1;
    }
}

class A
{
    public static int X = Test.F("Init A");
}

class B
{
    public static int Y = Test.F("Init B");
}
```

might produce either the output:

```
Init A
Init B
1 1
```

or the output:

```
Init B
Init A
1 1
```

because the execution of `x`'s initializer and `y`'s initializer could occur in either order; they are only constrained to occur before the references to those fields. However, in the example:

```

using System;

class Test
{
    static void Main() {
        Console.WriteLine("{0} {1}", B.Y, A.X);
    }

    public static int F(string s) {
        Console.WriteLine(s);
        return 1;
    }
}

class A
{
    static A() {}

    public static int X = Test.F("Init A");
}

class B
{
    static B() {}

    public static int Y = Test.F("Init B");
}

```

the output must be:

```

Init B
Init A
1 1

```

because the rules for when static constructors execute (as defined in [Static constructors](#)) provide that `B`'s static constructor (and hence `B`'s static field initializers) must run before `A`'s static constructor and field initializers.

Instance field initialization

The instance field variable initializers of a class correspond to a sequence of assignments that are executed immediately upon entry to any one of the instance constructors ([Constructor initializers](#)) of that class. The variable initializers are executed in the textual order in which they appear in the class declaration. The class instance creation and initialization process is described further in [Instance constructors](#).

A variable initializer for an instance field cannot reference the instance being created. Thus, it is a compile-time error to reference `this` in a variable initializer, as it is a compile-time error for a variable initializer to reference any instance member through a *simple_name*. In the example

```

class A
{
    int x = 1;
    int y = x + 1;    // Error, reference to instance member of this
}

```

the variable initializer for `y` results in a compile-time error because it references a member of the instance being created.

Methods

A *method* is a member that implements a computation or action that can be performed by an object or class.

Methods are declared using *method_declarations*:

```
method_declaration
  : method_header method_body
  ;

method_header
  : attributes? method_modifier* 'partial'? return_type member_name type_parameter_list?
    '(' formal_parameter_list? ')' type_parameter_constraints_clause*
  ;

method_modifier
  : 'new'
  | 'public'
  | 'protected'
  | 'internal'
  | 'private'
  | 'static'
  | 'virtual'
  | 'sealed'
  | 'override'
  | 'abstract'
  | 'extern'
  | 'async'
  | method_modifier_unsafe
  ;

return_type
  : type
  | 'void'
  ;

member_name
  : identifier
  | interface_type '.' identifier
  ;

method_body
  : block
  | '=>' expression ';'
  | ';'
  ;
```

A *method_declaration* may include a set of *attributes* ([Attributes](#)) and a valid combination of the four access modifiers ([Access modifiers](#)), the `new` ([The new modifier](#)), `static` ([Static and instance methods](#)), `virtual` ([Virtual methods](#)), `override` ([Override methods](#)), `sealed` ([Sealed methods](#)), `abstract` ([Abstract methods](#)), and `extern` ([External methods](#)) modifiers.

A declaration has a valid combination of modifiers if all of the following are true:

- The declaration includes a valid combination of access modifiers ([Access modifiers](#)).
- The declaration does not include the same modifier multiple times.
- The declaration includes at most one of the following modifiers: `static`, `virtual`, and `override`.
- The declaration includes at most one of the following modifiers: `new` and `override`.
- If the declaration includes the `abstract` modifier, then the declaration does not include any of the following modifiers: `static`, `virtual`, `sealed` or `extern`.
- If the declaration includes the `private` modifier, then the declaration does not include any of the following modifiers: `virtual`, `override`, or `abstract`.
- If the declaration includes the `sealed` modifier, then the declaration also includes the `override` modifier.
- If the declaration includes the `partial` modifier, then it does not include any of the following modifiers: `new`,

`public`, `protected`, `internal`, `private`, `virtual`, `sealed`, `override`, `abstract`, Or `extern`.

A method that has the `async` modifier is an async function and follows the rules described in [Async functions](#).

The *return_type* of a method declaration specifies the type of the value computed and returned by the method. The *return_type* is `void` if the method does not return a value. If the declaration includes the `partial` modifier, then the return type must be `void`.

The *member_name* specifies the name of the method. Unless the method is an explicit interface member implementation ([Explicit interface member implementations](#)), the *member_name* is simply an *identifier*. For an explicit interface member implementation, the *member_name* consists of an *interface_type* followed by a `"."` and an *identifier*.

The optional *type_parameter_list* specifies the type parameters of the method ([Type parameters](#)). If a *type_parameter_list* is specified the method is a **generic method**. If the method has an `extern` modifier, a *type_parameter_list* cannot be specified.

The optional *formal_parameter_list* specifies the parameters of the method ([Method parameters](#)).

The optional *type_parameter_constraints_clauses* specify constraints on individual type parameters ([Type parameter constraints](#)) and may only be specified if a *type_parameter_list* is also supplied, and the method does not have an `override` modifier.

The *return_type* and each of the types referenced in the *formal_parameter_list* of a method must be at least as accessible as the method itself ([Accessibility constraints](#)).

The *method_body* is either a semicolon, a **statement body** or an **expression body**. A statement body consists of a *block*, which specifies the statements to execute when the method is invoked. An expression body consists of `=>` followed by an *expression* and a semicolon, and denotes a single expression to perform when the method is invoked.

For `abstract` and `extern` methods, the *method_body* consists simply of a semicolon. For `partial` methods the *method_body* may consist of either a semicolon, a block body or an expression body. For all other methods, the *method_body* is either a block body or an expression body.

If the *method_body* consists of a semicolon, then the declaration may not include the `async` modifier.

The name, the type parameter list and the formal parameter list of a method define the signature ([Signatures and overloading](#)) of the method. Specifically, the signature of a method consists of its name, the number of type parameters and the number, modifiers, and types of its formal parameters. For these purposes, any type parameter of the method that occurs in the type of a formal parameter is identified not by its name, but by its ordinal position in the type argument list of the method. The return type is not part of a method's signature, nor are the names of the type parameters or the formal parameters.

The name of a method must differ from the names of all other non-methods declared in the same class. In addition, the signature of a method must differ from the signatures of all other methods declared in the same class, and two methods declared in the same class may not have signatures that differ solely by `ref` and `out`.

The method's *type_parameters* are in scope throughout the *method_declaration*, and can be used to form types throughout that scope in *return_type*, *method_body*, and *type_parameter_constraints_clauses* but not in *attributes*.

All formal parameters and type parameters must have different names.

Method parameters

The parameters of a method, if any, are declared by the method's *formal_parameter_list*.

```

formal_parameter_list
    : fixed_parameters
    | fixed_parameters ',' parameter_array
    | parameter_array
    ;

fixed_parameters
    : fixed_parameter (',' fixed_parameter)*
    ;

fixed_parameter
    : attributes? parameter_modifier? type identifier default_argument?
    ;

default_argument
    : '=' expression
    ;

parameter_modifier
    : 'ref'
    | 'out'
    | 'this'
    ;

parameter_array
    : attributes? 'params' array_type identifier
    ;

```

The formal parameter list consists of one or more comma-separated parameters of which only the last may be a *parameter_array*.

A *fixed_parameter* consists of an optional set of *attributes* ([Attributes](#)), an optional `ref`, `out` or `this` modifier, a *type*, an *identifier* and an optional *default_argument*. Each *fixed_parameter* declares a parameter of the given type with the given name. The `this` modifier designates the method as an extension method and is only allowed on the first parameter of a static method. Extension methods are further described in [Extension methods](#).

A *fixed_parameter* with a *default_argument* is known as an **optional parameter**, whereas a *fixed_parameter* without a *default_argument* is a **required parameter**. A required parameter may not appear after an optional parameter in a *formal_parameter_list*.

A `ref` or `out` parameter cannot have a *default_argument*. The *expression* in a *default_argument* must be one of the following:

- a *constant_expression*
- an expression of the form `new S()` where `S` is a value type
- an expression of the form `default(S)` where `S` is a value type

The *expression* must be implicitly convertible by an identity or nullable conversion to the type of the parameter.

If optional parameters occur in an implementing partial method declaration ([Partial methods](#)), an explicit interface member implementation ([Explicit interface member implementations](#)) or in a single-parameter indexer declaration ([Indexers](#)) the compiler should give a warning, since these members can never be invoked in a way that permits arguments to be omitted.

A *parameter_array* consists of an optional set of *attributes* ([Attributes](#)), a `params` modifier, an *array_type*, and an *identifier*. A parameter array declares a single parameter of the given array type with the given name. The *array_type* of a parameter array must be a single-dimensional array type ([Array types](#)). In a method invocation, a parameter array permits either a single argument of the given array type to be specified, or it permits zero or more arguments of the array element type to be specified. Parameter arrays are described further in [Parameter](#)

arrays.

A *parameter_array* may occur after an optional parameter, but cannot have a default value -- the omission of arguments for a *parameter_array* would instead result in the creation of an empty array.

The following example illustrates different kinds of parameters:

```
public void M(  
    ref int    i,  
    decimal    d,  
    bool       b = false,  
    bool?      n = false,  
    string     s = "Hello",  
    object     o = null,  
    T          t = default(T),  
    params int[] a  
) { }
```

In the *formal_parameter_list* for `M`, `i` is a required ref parameter, `d` is a required value parameter, `b`, `s`, `o` and `t` are optional value parameters and `a` is a parameter array.

A method declaration creates a separate declaration space for parameters, type parameters and local variables. Names are introduced into this declaration space by the type parameter list and the formal parameter list of the method and by local variable declarations in the *block* of the method. It is an error for two members of a method declaration space to have the same name. It is an error for the method declaration space and the local variable declaration space of a nested declaration space to contain elements with the same name.

A method invocation ([Method invocations](#)) creates a copy, specific to that invocation, of the formal parameters and local variables of the method, and the argument list of the invocation assigns values or variable references to the newly created formal parameters. Within the *block* of a method, formal parameters can be referenced by their identifiers in *simple_name* expressions ([Simple names](#)).

There are four kinds of formal parameters:

- Value parameters, which are declared without any modifiers.
- Reference parameters, which are declared with the `ref` modifier.
- Output parameters, which are declared with the `out` modifier.
- Parameter arrays, which are declared with the `params` modifier.

As described in [Signatures and overloading](#), the `ref` and `out` modifiers are part of a method's signature, but the `params` modifier is not.

Value parameters

A parameter declared with no modifiers is a value parameter. A value parameter corresponds to a local variable that gets its initial value from the corresponding argument supplied in the method invocation.

When a formal parameter is a value parameter, the corresponding argument in a method invocation must be an expression that is implicitly convertible ([Implicit conversions](#)) to the formal parameter type.

A method is permitted to assign new values to a value parameter. Such assignments only affect the local storage location represented by the value parameter—they have no effect on the actual argument given in the method invocation.

Reference parameters

A parameter declared with a `ref` modifier is a reference parameter. Unlike a value parameter, a reference parameter does not create a new storage location. Instead, a reference parameter represents the same storage location as the variable given as the argument in the method invocation.

When a formal parameter is a reference parameter, the corresponding argument in a method invocation must consist of the keyword `ref` followed by a *variable_reference* ([Precise rules for determining definite assignment](#)) of the same type as the formal parameter. A variable must be definitely assigned before it can be passed as a reference parameter.

Within a method, a reference parameter is always considered definitely assigned.

A method declared as an iterator ([Iterators](#)) cannot have reference parameters.

The example

```
using System;

class Test
{
    static void Swap(ref int x, ref int y) {
        int temp = x;
        x = y;
        y = temp;
    }

    static void Main() {
        int i = 1, j = 2;
        Swap(ref i, ref j);
        Console.WriteLine("i = {0}, j = {1}", i, j);
    }
}
```

produces the output

```
i = 2, j = 1
```

For the invocation of `Swap` in `Main`, `x` represents `i` and `y` represents `j`. Thus, the invocation has the effect of swapping the values of `i` and `j`.

In a method that takes reference parameters it is possible for multiple names to represent the same storage location. In the example

```
class A
{
    string s;

    void F(ref string a, ref string b) {
        s = "One";
        a = "Two";
        b = "Three";
    }

    void G() {
        F(ref s, ref s);
    }
}
```

the invocation of `F` in `G` passes a reference to `s` for both `a` and `b`. Thus, for that invocation, the names `s`, `a`, and `b` all refer to the same storage location, and the three assignments all modify the instance field `s`.

Output parameters

A parameter declared with an `out` modifier is an output parameter. Similar to a reference parameter, an output parameter does not create a new storage location. Instead, an output parameter represents the same storage location as the variable given as the argument in the method invocation.

When a formal parameter is an output parameter, the corresponding argument in a method invocation must consist of the keyword `out` followed by a *variable_reference* ([Precise rules for determining definite assignment](#)) of the same type as the formal parameter. A variable need not be definitely assigned before it can be passed as an output parameter, but following an invocation where a variable was passed as an output parameter, the variable is considered definitely assigned.

Within a method, just like a local variable, an output parameter is initially considered unassigned and must be definitely assigned before its value is used.

Every output parameter of a method must be definitely assigned before the method returns.

A method declared as a partial method ([Partial methods](#)) or an iterator ([Iterators](#)) cannot have output parameters.

Output parameters are typically used in methods that produce multiple return values. For example:

```
using System;

class Test
{
    static void SplitPath(string path, out string dir, out string name) {
        int i = path.Length;
        while (i > 0) {
            char ch = path[i - 1];
            if (ch == '\\' || ch == '/' || ch == ':') break;
            i--;
        }
        dir = path.Substring(0, i);
        name = path.Substring(i);
    }

    static void Main() {
        string dir, name;
        SplitPath("c:\\Windows\\System\\hello.txt", out dir, out name);
        Console.WriteLine(dir);
        Console.WriteLine(name);
    }
}
```

The example produces the output:

```
c:\Windows\System\
hello.txt
```

Note that the `dir` and `name` variables can be unassigned before they are passed to `SplitPath`, and that they are considered definitely assigned following the call.

Parameter arrays

A parameter declared with a `params` modifier is a parameter array. If a formal parameter list includes a parameter array, it must be the last parameter in the list and it must be of a single-dimensional array type. For example, the types `string[]` and `string[][]` can be used as the type of a parameter array, but the type `string[,]` can not. It is not possible to combine the `params` modifier with the modifiers `ref` and `out`.

A parameter array permits arguments to be specified in one of two ways in a method invocation:

- The argument given for a parameter array can be a single expression that is implicitly convertible ([Implicit conversions](#)) to the parameter array type. In this case, the parameter array acts precisely like a value parameter.
- Alternatively, the invocation can specify zero or more arguments for the parameter array, where each

argument is an expression that is implicitly convertible ([Implicit conversions](#)) to the element type of the parameter array. In this case, the invocation creates an instance of the parameter array type with a length corresponding to the number of arguments, initializes the elements of the array instance with the given argument values, and uses the newly created array instance as the actual argument.

Except for allowing a variable number of arguments in an invocation, a parameter array is precisely equivalent to a value parameter ([Value parameters](#)) of the same type.

The example

```
using System;

class Test
{
    static void F(params int[] args) {
        Console.WriteLine("Array contains {0} elements:", args.Length);
        foreach (int i in args)
            Console.WriteLine(" {0}", i);
        Console.WriteLine();
    }

    static void Main() {
        int[] arr = {1, 2, 3};
        F(arr);
        F(10, 20, 30, 40);
        F();
    }
}
```

produces the output

```
Array contains 3 elements: 1 2 3
Array contains 4 elements: 10 20 30 40
Array contains 0 elements:
```

The first invocation of `F` simply passes the array `a` as a value parameter. The second invocation of `F` automatically creates a four-element `int[]` with the given element values and passes that array instance as a value parameter. Likewise, the third invocation of `F` creates a zero-element `int[]` and passes that instance as a value parameter. The second and third invocations are precisely equivalent to writing:

```
F(new int[] {10, 20, 30, 40});
F(new int[] {});
```

When performing overload resolution, a method with a parameter array may be applicable either in its normal form or in its expanded form ([Applicable function member](#)). The expanded form of a method is available only if the normal form of the method is not applicable and only if an applicable method with the same signature as the expanded form is not already declared in the same type.

The example

```

using System;

class Test
{
    static void F(params object[] a) {
        Console.WriteLine("F(object[])");
    }

    static void F() {
        Console.WriteLine("F()");
    }

    static void F(object a0, object a1) {
        Console.WriteLine("F(object,object)");
    }

    static void Main() {
        F();
        F(1);
        F(1, 2);
        F(1, 2, 3);
        F(1, 2, 3, 4);
    }
}

```

produces the output

```

F();
F(object[]);
F(object,object);
F(object[]);
F(object[]);

```

In the example, two of the possible expanded forms of the method with a parameter array are already included in the class as regular methods. These expanded forms are therefore not considered when performing overload resolution, and the first and third method invocations thus select the regular methods. When a class declares a method with a parameter array, it is not uncommon to also include some of the expanded forms as regular methods. By doing so it is possible to avoid the allocation of an array instance that occurs when an expanded form of a method with a parameter array is invoked.

When the type of a parameter array is `object[]`, a potential ambiguity arises between the normal form of the method and the expanded form for a single `object` parameter. The reason for the ambiguity is that an `object[]` is itself implicitly convertible to type `object`. The ambiguity presents no problem, however, since it can be resolved by inserting a cast if needed.

The example

```

using System;

class Test
{
    static void F(params object[] args) {
        foreach (object o in args) {
            Console.Write(o.GetType().FullName);
            Console.Write(" ");
        }
        Console.WriteLine();
    }

    static void Main() {
        object[] a = {1, "Hello", 123.456};
        object o = a;
        F(a);
        F((object)a);
        F(o);
        F((object[])o);
    }
}

```

produces the output

```

System.Int32 System.String System.Double
System.Object[]
System.Object[]
System.Int32 System.String System.Double

```

In the first and last invocations of `F`, the normal form of `F` is applicable because an implicit conversion exists from the argument type to the parameter type (both are of type `object[]`). Thus, overload resolution selects the normal form of `F`, and the argument is passed as a regular value parameter. In the second and third invocations, the normal form of `F` is not applicable because no implicit conversion exists from the argument type to the parameter type (type `object` cannot be implicitly converted to type `object[]`). However, the expanded form of `F` is applicable, so it is selected by overload resolution. As a result, a one-element `object[]` is created by the invocation, and the single element of the array is initialized with the given argument value (which itself is a reference to an `object[]`).

Static and instance methods

When a method declaration includes a `static` modifier, that method is said to be a static method. When no `static` modifier is present, the method is said to be an instance method.

A static method does not operate on a specific instance, and it is a compile-time error to refer to `this` in a static method.

An instance method operates on a given instance of a class, and that instance can be accessed as `this` ([This access](#)).

When a method is referenced in a *member_access* ([Member access](#)) of the form `E.M`, if `M` is a static method, `E` must denote a type containing `M`, and if `M` is an instance method, `E` must denote an instance of a type containing `M`.

The differences between static and instance members are discussed further in [Static and instance members](#).

Virtual methods

When an instance method declaration includes a `virtual` modifier, that method is said to be a virtual method. When no `virtual` modifier is present, the method is said to be a non-virtual method.

The implementation of a non-virtual method is invariant: The implementation is the same whether the method is invoked on an instance of the class in which it is declared or an instance of a derived class. In contrast, the implementation of a virtual method can be superseded by derived classes. The process of superseding the implementation of an inherited virtual method is known as *overriding* that method ([Override methods](#)).

In a virtual method invocation, the *run-time type* of the instance for which that invocation takes place determines the actual method implementation to invoke. In a non-virtual method invocation, the *compile-time type* of the instance is the determining factor. In precise terms, when a method named `M` is invoked with an argument list `A` on an instance with a compile-time type `C` and a run-time type `R` (where `R` is either `C` or a class derived from `C`), the invocation is processed as follows:

- First, overload resolution is applied to `C`, `M`, and `A`, to select a specific method `M` from the set of methods declared in and inherited by `C`. This is described in [Method invocations](#).
- Then, if `M` is a non-virtual method, `M` is invoked.
- Otherwise, `M` is a virtual method, and the most derived implementation of `M` with respect to `R` is invoked.

For every virtual method declared in or inherited by a class, there exists a *most derived implementation* of the method with respect to that class. The most derived implementation of a virtual method `M` with respect to a class `R` is determined as follows:

- If `R` contains the introducing `virtual` declaration of `M`, then this is the most derived implementation of `M`.
- Otherwise, if `R` contains an `override` of `M`, then this is the most derived implementation of `M`.
- Otherwise, the most derived implementation of `M` with respect to `R` is the same as the most derived implementation of `M` with respect to the direct base class of `R`.

The following example illustrates the differences between virtual and non-virtual methods:

```
using System;

class A
{
    public void F() { Console.WriteLine("A.F"); }

    public virtual void G() { Console.WriteLine("A.G"); }
}

class B : A
{
    new public void F() { Console.WriteLine("B.F"); }

    public override void G() { Console.WriteLine("B.G"); }
}

class Test
{
    static void Main() {
        B b = new B();
        A a = b;
        a.F();
        b.F();
        a.G();
        b.G();
    }
}
```

In the example, `A` introduces a non-virtual method `F` and a virtual method `G`. The class `B` introduces a new non-virtual method `F`, thus hiding the inherited `F`, and also overrides the inherited method `G`. The example produces the output:

A.F
B.F
B.G
B.G

Notice that the statement `a.G()` invokes `B.G`, not `A.G`. This is because the run-time type of the instance (which is `B`), not the compile-time type of the instance (which is `A`), determines the actual method implementation to invoke.

Because methods are allowed to hide inherited methods, it is possible for a class to contain several virtual methods with the same signature. This does not present an ambiguity problem, since all but the most derived method are hidden. In the example

```
using System;

class A
{
    public virtual void F() { Console.WriteLine("A.F"); }
}

class B: A
{
    public override void F() { Console.WriteLine("B.F"); }
}

class C: B
{
    new public virtual void F() { Console.WriteLine("C.F"); }
}

class D: C
{
    public override void F() { Console.WriteLine("D.F"); }
}

class Test
{
    static void Main() {
        D d = new D();
        A a = d;
        B b = d;
        C c = d;
        a.F();
        b.F();
        c.F();
        d.F();
    }
}
```

the `C` and `D` classes contain two virtual methods with the same signature: The one introduced by `A` and the one introduced by `C`. The method introduced by `C` hides the method inherited from `A`. Thus, the override declaration in `D` overrides the method introduced by `C`, and it is not possible for `D` to override the method introduced by `A`. The example produces the output:

B.F
B.F
D.F
D.F

Note that it is possible to invoke the hidden virtual method by accessing an instance of `D` through a less

derived type in which the method is not hidden.

Override methods

When an instance method declaration includes an `override` modifier, the method is said to be an **override method**. An override method overrides an inherited virtual method with the same signature. Whereas a virtual method declaration introduces a new method, an override method declaration specializes an existing inherited virtual method by providing a new implementation of that method.

The method overridden by an `override` declaration is known as the **overridden base method**. For an override method `M` declared in a class `C`, the overridden base method is determined by examining each base class type of `C`, starting with the direct base class type of `C` and continuing with each successive direct base class type, until in a given base class type at least one accessible method is located which has the same signature as `M` after substitution of type arguments. For the purposes of locating the overridden base method, a method is considered accessible if it is `public`, if it is `protected`, if it is `protected internal`, or if it is `internal` and declared in the same program as `C`.

A compile-time error occurs unless all of the following are true for an override declaration:

- An overridden base method can be located as described above.
- There is exactly one such overridden base method. This restriction has effect only if the base class type is a constructed type where the substitution of type arguments makes the signature of two methods the same.
- The overridden base method is a virtual, abstract, or override method. In other words, the overridden base method cannot be static or non-virtual.
- The overridden base method is not a sealed method.
- The override method and the overridden base method have the same return type.
- The override declaration and the overridden base method have the same declared accessibility. In other words, an override declaration cannot change the accessibility of the virtual method. However, if the overridden base method is `protected internal` and it is declared in a different assembly than the assembly containing the override method then the override method's declared accessibility must be `protected`.
- The override declaration does not specify type-parameter-constraints-clauses. Instead the constraints are inherited from the overridden base method. Note that constraints that are type parameters in the overridden method may be replaced by type arguments in the inherited constraint. This can lead to constraints that are not legal when explicitly specified, such as value types or sealed types.

The following example demonstrates how the overriding rules work for generic classes:

```
abstract class C<T>
{
    public virtual T F() {...}
    public virtual C<T> G() {...}
    public virtual void H(C<T> x) {...}
}

class D: C<string>
{
    public override string F() {...}           // Ok
    public override C<string> G() {...}        // Ok
    public override void H(C<T> x) {...}       // Error, should be C<string>
}

class E<T,U>: C<U>
{
    public override U F() {...}               // Ok
    public override C<U> G() {...}            // Ok
    public override void H(C<T> x) {...}       // Error, should be C<U>
}
```

An override declaration can access the overridden base method using a *base_access* (Base access). In the example

```
class A
{
    int x;

    public virtual void PrintFields() {
        Console.WriteLine("x = {0}", x);
    }
}

class B: A
{
    int y;

    public override void PrintFields() {
        base.PrintFields();
        Console.WriteLine("y = {0}", y);
    }
}
```

the `base.PrintFields()` invocation in `B` invokes the `PrintFields` method declared in `A`. A *base_access* disables the virtual invocation mechanism and simply treats the base method as a non-virtual method. Had the invocation in `B` been written `((A)this).PrintFields()`, it would recursively invoke the `PrintFields` method declared in `B`, not the one declared in `A`, since `PrintFields` is virtual and the run-time type of `((A)this)` is `B`.

Only by including an `override` modifier can a method override another method. In all other cases, a method with the same signature as an inherited method simply hides the inherited method. In the example

```
class A
{
    public virtual void F() {}
}

class B: A
{
    public virtual void F() {}    // Warning, hiding inherited F()
}
```

the `F` method in `B` does not include an `override` modifier and therefore does not override the `F` method in `A`. Rather, the `F` method in `B` hides the method in `A`, and a warning is reported because the declaration does not include a `new` modifier.

In the example

```

class A
{
    public virtual void F() {}
}

class B: A
{
    new private void F() {}      // Hides A.F within body of B
}

class C: B
{
    public override void F() {}  // Ok, overrides A.F
}

```

the `F` method in `B` hides the virtual `F` method inherited from `A`. Since the new `F` in `B` has private access, its scope only includes the class body of `B` and does not extend to `C`. Therefore, the declaration of `F` in `C` is permitted to override the `F` inherited from `A`.

Sealed methods

When an instance method declaration includes a `sealed` modifier, that method is said to be a *sealed method*. If an instance method declaration includes the `sealed` modifier, it must also include the `override` modifier. Use of the `sealed` modifier prevents a derived class from further overriding the method.

In the example

```

using System;

class A
{
    public virtual void F() {
        Console.WriteLine("A.F");
    }

    public virtual void G() {
        Console.WriteLine("A.G");
    }
}

class B: A
{
    sealed override public void F() {
        Console.WriteLine("B.F");
    }

    override public void G() {
        Console.WriteLine("B.G");
    }
}

class C: B
{
    override public void G() {
        Console.WriteLine("C.G");
    }
}

```

the class `B` provides two override methods: an `F` method that has the `sealed` modifier and a `G` method that does not. `B`'s use of the `sealed` modifier prevents `C` from further overriding `F`.

Abstract methods

When an instance method declaration includes an `abstract` modifier, that method is said to be an ***abstract method***. Although an abstract method is implicitly also a virtual method, it cannot have the modifier `virtual`.

An abstract method declaration introduces a new virtual method but does not provide an implementation of that method. Instead, non-abstract derived classes are required to provide their own implementation by overriding that method. Because an abstract method provides no actual implementation, the *method_body* of an abstract method simply consists of a semicolon.

Abstract method declarations are only permitted in abstract classes ([Abstract classes](#)).

In the example

```
public abstract class Shape
{
    public abstract void Paint(Graphics g, Rectangle r);
}

public class Ellipse: Shape
{
    public override void Paint(Graphics g, Rectangle r) {
        g.DrawEllipse(r);
    }
}

public class Box: Shape
{
    public override void Paint(Graphics g, Rectangle r) {
        g.DrawRect(r);
    }
}
```

the `Shape` class defines the abstract notion of a geometrical shape object that can paint itself. The `Paint` method is abstract because there is no meaningful default implementation. The `Ellipse` and `Box` classes are concrete `Shape` implementations. Because these classes are non-abstract, they are required to override the `Paint` method and provide an actual implementation.

It is a compile-time error for a *base_access* ([Base access](#)) to reference an abstract method. In the example

```
abstract class A
{
    public abstract void F();
}

class B: A
{
    public override void F() {
        base.F(); // Error, base.F is abstract
    }
}
```

a compile-time error is reported for the `base.F()` invocation because it references an abstract method.

An abstract method declaration is permitted to override a virtual method. This allows an abstract class to force re-implementation of the method in derived classes, and makes the original implementation of the method unavailable. In the example

```

using System;

class A
{
    public virtual void F() {
        Console.WriteLine("A.F");
    }
}

abstract class B: A
{
    public abstract override void F();
}

class C: B
{
    public override void F() {
        Console.WriteLine("C.F");
    }
}

```

class `A` declares a virtual method, class `B` overrides this method with an abstract method, and class `C` overrides the abstract method to provide its own implementation.

External methods

When a method declaration includes an `extern` modifier, that method is said to be an *external method*. External methods are implemented externally, typically using a language other than C#. Because an external method declaration provides no actual implementation, the *method body* of an external method simply consists of a semicolon. An external method may not be generic.

The `extern` modifier is typically used in conjunction with a `DllImport` attribute ([Interoperation with COM and Win32 components](#)), allowing external methods to be implemented by DLLs (Dynamic Link Libraries). The execution environment may support other mechanisms whereby implementations of external methods can be provided.

When an external method includes a `DllImport` attribute, the method declaration must also include a `static` modifier. This example demonstrates the use of the `extern` modifier and the `DllImport` attribute:

```

using System.Text;
using System.Security.Permissions;
using System.Runtime.InteropServices;

class Path
{
    [DllImport("kernel32", SetLastError=true)]
    static extern bool CreateDirectory(string name, SecurityAttribute sa);

    [DllImport("kernel32", SetLastError=true)]
    static extern bool RemoveDirectory(string name);

    [DllImport("kernel32", SetLastError=true)]
    static extern int GetCurrentDirectory(int bufSize, StringBuilder buf);

    [DllImport("kernel32", SetLastError=true)]
    static extern bool SetCurrentDirectory(string name);
}

```

Partial methods (recap)

When a method declaration includes a `partial` modifier, that method is said to be a *partial method*. Partial methods can only be declared as members of partial types ([Partial types](#)), and are subject to a number of

restrictions. Partial methods are further described in [Partial methods](#).

Extension methods

When the first parameter of a method includes the `this` modifier, that method is said to be an *extension method*. Extension methods can only be declared in non-generic, non-nested static classes. The first parameter of an extension method can have no modifiers other than `this`, and the parameter type cannot be a pointer type.

The following is an example of a static class that declares two extension methods:

```
public static class Extensions
{
    public static int ToInt32(this string s) {
        return Int32.Parse(s);
    }

    public static T[] Slice<T>(this T[] source, int index, int count) {
        if (index < 0 || count < 0 || source.Length - index < count)
            throw new ArgumentException();
        T[] result = new T[count];
        Array.Copy(source, index, result, 0, count);
        return result;
    }
}
```

An extension method is a regular static method. In addition, where its enclosing static class is in scope, an extension method can be invoked using instance method invocation syntax ([Extension method invocations](#)), using the receiver expression as the first argument.

The following program uses the extension methods declared above:

```
static class Program
{
    static void Main() {
        string[] strings = { "1", "22", "333", "4444" };
        foreach (string s in strings.Slice(1, 2)) {
            Console.WriteLine(s.ToInt32());
        }
    }
}
```

The `slice` method is available on the `string[]`, and the `ToInt32` method is available on `string`, because they have been declared as extension methods. The meaning of the program is the same as the following, using ordinary static method calls:

```
static class Program
{
    static void Main() {
        string[] strings = { "1", "22", "333", "4444" };
        foreach (string s in Extensions.Slice(strings, 1, 2)) {
            Console.WriteLine(Extensions.ToInt32(s));
        }
    }
}
```

Method body

The *method body* of a method declaration consists of either a block body, an expression body or a semicolon.

The *result type* of a method is `void` if the return type is `void`, or if the method is `async` and the return type is

`System.Threading.Tasks.Task`. Otherwise, the result type of a non-async method is its return type, and the result type of an async method with return type `System.Threading.Tasks.Task<T>` is `T`.

When a method has a `void` result type and a block body, `return` statements ([The return statement](#)) in the block are not permitted to specify an expression. If execution of the block of a void method completes normally (that is, control flows off the end of the method body), that method simply returns to its current caller.

When a method has a `void` result and an expression body, the expression `E` must be a *statement_expression*, and the body is exactly equivalent to a block body of the form `{ E; }`.

When a method has a non-void result type and a block body, each `return` statement in the block must specify an expression that is implicitly convertible to the result type. The endpoint of a block body of a value-returning method must not be reachable. In other words, in a value-returning method with a block body, control is not permitted to flow off the end of the method body.

When a method has a non-void result type and an expression body, the expression must be implicitly convertible to the result type, and the body is exactly equivalent to a block body of the form `{ return E; }`.

In the example

```
class A
{
    public int F() {}           // Error, return value required

    public int G() {
        return 1;
    }

    public int H(bool b) {
        if (b) {
            return 1;
        }
        else {
            return 0;
        }
    }

    public int I(bool b) => b ? 1 : 0;
}
```

the value-returning `F` method results in a compile-time error because control can flow off the end of the method body. The `G` and `H` methods are correct because all possible execution paths end in a return statement that specifies a return value. The `I` method is correct, because its body is equivalent to a statement block with just a single return statement in it.

Method overloading

The method overload resolution rules are described in [Type inference](#).

Properties

A *property* is a member that provides access to a characteristic of an object or a class. Examples of properties include the length of a string, the size of a font, the caption of a window, the name of a customer, and so on. Properties are a natural extension of fields—both are named members with associated types, and the syntax for accessing fields and properties is the same. However, unlike fields, properties do not denote storage locations. Instead, properties have *accessors* that specify the statements to be executed when their values are read or written. Properties thus provide a mechanism for associating actions with the reading and writing of an object's attributes; furthermore, they permit such attributes to be computed.

Properties are declared using *property_declarations*:

```

property_declaration
    : attributes? property_modifier* type member_name property_body
    ;

property_modifier
    : 'new'
    | 'public'
    | 'protected'
    | 'internal'
    | 'private'
    | 'static'
    | 'virtual'
    | 'sealed'
    | 'override'
    | 'abstract'
    | 'extern'
    | property_modifier_unsafe
    ;

property_body
    : '{' accessor_declarations '}' property_initializer?
    | '=>' expression ';'
    ;

property_initializer
    : '=' variable_initializer ';'
    ;

```

A *property_declaration* may include a set of *attributes* ([Attributes](#)) and a valid combination of the four access modifiers ([Access modifiers](#)), the `new` ([The new modifier](#)), `static` ([Static and instance methods](#)), `virtual` ([Virtual methods](#)), `override` ([Override methods](#)), `sealed` ([Sealed methods](#)), `abstract` ([Abstract methods](#)), and `extern` ([External methods](#)) modifiers.

Property declarations are subject to the same rules as method declarations ([Methods](#)) with regard to valid combinations of modifiers.

The *type* of a property declaration specifies the type of the property introduced by the declaration, and the *member_name* specifies the name of the property. Unless the property is an explicit interface member implementation, the *member_name* is simply an *identifier*. For an explicit interface member implementation ([Explicit interface member implementations](#)), the *member_name* consists of an *interface_type* followed by a `"."` and an *identifier*.

The *type* of a property must be at least as accessible as the property itself ([Accessibility constraints](#)).

A *property_body* may either consist of an *accessor body* or an *expression body*. In an accessor body, *accessor_declarations*, which must be enclosed in `"{"` and `"}"` tokens, declare the accessors ([Accessors](#)) of the property. The accessors specify the executable statements associated with reading and writing the property.

An expression body consisting of `=>` followed by an *expression* `E` and a semicolon is exactly equivalent to the statement body `{ get { return E; } }`, and can therefore only be used to specify getter-only properties where the result of the getter is given by a single expression.

A *property_initializer* may only be given for an automatically implemented property ([Automatically implemented properties](#)), and causes the initialization of the underlying field of such properties with the value given by the *expression*.

Even though the syntax for accessing a property is the same as that for a field, a property is not classified as a variable. Thus, it is not possible to pass a property as a `ref` or `out` argument.

When a property declaration includes an `extern` modifier, the property is said to be an *external property*.

Because an external property declaration provides no actual implementation, each of its *accessor_declarations* consists of a semicolon.

Static and instance properties

When a property declaration includes a `static` modifier, the property is said to be a *static property*. When no `static` modifier is present, the property is said to be an *instance property*.

A static property is not associated with a specific instance, and it is a compile-time error to refer to `this` in the accessors of a static property.

An instance property is associated with a given instance of a class, and that instance can be accessed as `this` ([This access](#)) in the accessors of that property.

When a property is referenced in a *member_access* ([Member access](#)) of the form `E.M`, if `M` is a static property, `E` must denote a type containing `M`, and if `M` is an instance property, `E` must denote an instance of a type containing `M`.

The differences between static and instance members are discussed further in [Static and instance members](#).

Accessors

The *accessor_declarations* of a property specify the executable statements associated with reading and writing that property.

```
accessor_declarations
: get_accessor_declaration set_accessor_declaration?
| set_accessor_declaration get_accessor_declaration?
;

get_accessor_declaration
: attributes? accessor_modifier? 'get' accessor_body
;

set_accessor_declaration
: attributes? accessor_modifier? 'set' accessor_body
;

accessor_modifier
: 'protected'
| 'internal'
| 'private'
| 'protected' 'internal'
| 'internal' 'protected'
;

accessor_body
: block
| ';'
;
```

The accessor declarations consist of a *get_accessor_declaration*, a *set_accessor_declaration*, or both. Each accessor declaration consists of the token `get` or `set` followed by an optional *accessor_modifier* and an *accessor_body*.

The use of *accessor_modifiers* is governed by the following restrictions:

- An *accessor_modifier* may not be used in an interface or in an explicit interface member implementation.
- For a property or indexer that has no `override` modifier, an *accessor_modifier* is permitted only if the property or indexer has both a `get` and `set` accessor, and then is permitted only on one of those accessors.
- For a property or indexer that includes an `override` modifier, an accessor must match the *accessor_modifier*, if any, of the accessor being overridden.

- The *accessor_modifier* must declare an accessibility that is strictly more restrictive than the declared accessibility of the property or indexer itself. To be precise:
 - If the property or indexer has a declared accessibility of `public`, the *accessor_modifier* may be either `protected internal`, `internal`, `protected`, or `private`.
 - If the property or indexer has a declared accessibility of `protected internal`, the *accessor_modifier* may be either `internal`, `protected`, or `private`.
 - If the property or indexer has a declared accessibility of `internal` or `protected`, the *accessor_modifier* must be `private`.
 - If the property or indexer has a declared accessibility of `private`, no *accessor_modifier* may be used.

For `abstract` and `extern` properties, the *accessor_body* for each accessor specified is simply a semicolon. A non-abstract, non-extern property may have each *accessor_body* be a semicolon, in which case it is an **automatically implemented property** ([Automatically implemented properties](#)). An automatically implemented property must have at least a get accessor. For the accessors of any other non-abstract, non-extern property, the *accessor_body* is a *block* which specifies the statements to be executed when the corresponding accessor is invoked.

A `get` accessor corresponds to a parameterless method with a return value of the property type. Except as the target of an assignment, when a property is referenced in an expression, the `get` accessor of the property is invoked to compute the value of the property ([Values of expressions](#)). The body of a `get` accessor must conform to the rules for value-returning methods described in [Method body](#). In particular, all `return` statements in the body of a `get` accessor must specify an expression that is implicitly convertible to the property type. Furthermore, the endpoint of a `get` accessor must not be reachable.

A `set` accessor corresponds to a method with a single value parameter of the property type and a `void` return type. The implicit parameter of a `set` accessor is always named `value`. When a property is referenced as the target of an assignment ([Assignment operators](#)), or as the operand of `++` or `--` ([Postfix increment and decrement operators](#), [Prefix increment and decrement operators](#)), the `set` accessor is invoked with an argument (whose value is that of the right-hand side of the assignment or the operand of the `++` or `--` operator) that provides the new value ([Simple assignment](#)). The body of a `set` accessor must conform to the rules for `void` methods described in [Method body](#). In particular, `return` statements in the `set` accessor body are not permitted to specify an expression. Since a `set` accessor implicitly has a parameter named `value`, it is a compile-time error for a local variable or constant declaration in a `set` accessor to have that name.

Based on the presence or absence of the `get` and `set` accessors, a property is classified as follows:

- A property that includes both a `get` accessor and a `set` accessor is said to be a **read-write** property.
- A property that has only a `get` accessor is said to be a **read-only** property. It is a compile-time error for a read-only property to be the target of an assignment.
- A property that has only a `set` accessor is said to be a **write-only** property. Except as the target of an assignment, it is a compile-time error to reference a write-only property in an expression.

In the example

```

public class Button: Control
{
    private string caption;

    public string Caption {
        get {
            return caption;
        }
        set {
            if (caption != value) {
                caption = value;
                Repaint();
            }
        }
    }

    public override void Paint(Graphics g, Rectangle r) {
        // Painting code goes here
    }
}

```

the `Button` control declares a public `Caption` property. The `get` accessor of the `Caption` property returns the string stored in the private `caption` field. The `set` accessor checks if the new value is different from the current value, and if so, it stores the new value and repaints the control. Properties often follow the pattern shown above: The `get` accessor simply returns a value stored in a private field, and the `set` accessor modifies that private field and then performs any additional actions required to fully update the state of the object.

Given the `Button` class above, the following is an example of use of the `Caption` property:

```

Button okButton = new Button();
okButton.Caption = "OK";           // Invokes set accessor
string s = okButton.Caption;       // Invokes get accessor

```

Here, the `set` accessor is invoked by assigning a value to the property, and the `get` accessor is invoked by referencing the property in an expression.

The `get` and `set` accessors of a property are not distinct members, and it is not possible to declare the accessors of a property separately. As such, it is not possible for the two accessors of a read-write property to have different accessibility. The example

```

class A
{
    private string name;

    public string Name {                // Error, duplicate member name
        get { return name; }
    }

    public string Name {                // Error, duplicate member name
        set { name = value; }
    }
}

```

does not declare a single read-write property. Rather, it declares two properties with the same name, one read-only and one write-only. Since two members declared in the same class cannot have the same name, the example causes a compile-time error to occur.

When a derived class declares a property by the same name as an inherited property, the derived property hides the inherited property with respect to both reading and writing. In the example

```

class A
{
    public int P {
        set {...}
    }
}

class B: A
{
    new public int P {
        get {...}
    }
}

```

the `P` property in `B` hides the `P` property in `A` with respect to both reading and writing. Thus, in the statements

```

B b = new B();
b.P = 1;           // Error, B.P is read-only
((A)b).P = 1;      // Ok, reference to A.P

```

the assignment to `b.P` causes a compile-time error to be reported, since the read-only `P` property in `B` hides the write-only `P` property in `A`. Note, however, that a cast can be used to access the hidden `P` property.

Unlike public fields, properties provide a separation between an object's internal state and its public interface. Consider the example:

```

class Label
{
    private int x, y;
    private string caption;

    public Label(int x, int y, string caption) {
        this.x = x;
        this.y = y;
        this.caption = caption;
    }

    public int X {
        get { return x; }
    }

    public int Y {
        get { return y; }
    }

    public Point Location {
        get { return new Point(x, y); }
    }

    public string Caption {
        get { return caption; }
    }
}

```

Here, the `Label` class uses two `int` fields, `x` and `y`, to store its location. The location is publicly exposed both as an `X` and a `Y` property and as a `Location` property of type `Point`. If, in a future version of `Label`, it becomes more convenient to store the location as a `Point` internally, the change can be made without affecting the public interface of the class:

```

class Label
{
    private Point location;
    private string caption;

    public Label(int x, int y, string caption) {
        this.location = new Point(x, y);
        this.caption = caption;
    }

    public int X {
        get { return location.x; }
    }

    public int Y {
        get { return location.y; }
    }

    public Point Location {
        get { return location; }
    }

    public string Caption {
        get { return caption; }
    }
}

```

Had `x` and `y` instead been `public readonly` fields, it would have been impossible to make such a change to the `Label` class.

Exposing state through properties is not necessarily any less efficient than exposing fields directly. In particular, when a property is non-virtual and contains only a small amount of code, the execution environment may replace calls to accessors with the actual code of the accessors. This process is known as *inlining*, and it makes property access as efficient as field access, yet preserves the increased flexibility of properties.

Since invoking a `get` accessor is conceptually equivalent to reading the value of a field, it is considered bad programming style for `get` accessors to have observable side-effects. In the example

```

class Counter
{
    private int next;

    public int Next {
        get { return next++; }
    }
}

```

the value of the `Next` property depends on the number of times the property has previously been accessed. Thus, accessing the property produces an observable side-effect, and the property should be implemented as a method instead.

The "no side-effects" convention for `get` accessors doesn't mean that `get` accessors should always be written to simply return values stored in fields. Indeed, `get` accessors often compute the value of a property by accessing multiple fields or invoking methods. However, a properly designed `get` accessor performs no actions that cause observable changes in the state of the object.

Properties can be used to delay initialization of a resource until the moment it is first referenced. For example:

```

using System.IO;

public class Console
{
    private static TextReader reader;
    private static TextWriter writer;
    private static TextWriter error;

    public static TextReader In {
        get {
            if (reader == null) {
                reader = new StreamReader(Console.OpenStandardInput());
            }
            return reader;
        }
    }

    public static TextWriter Out {
        get {
            if (writer == null) {
                writer = new StreamWriter(Console.OpenStandardOutput());
            }
            return writer;
        }
    }

    public static TextWriter Error {
        get {
            if (error == null) {
                error = new StreamWriter(Console.OpenStandardError());
            }
            return error;
        }
    }
}

```

The `Console` class contains three properties, `In`, `Out`, and `Error`, that represent the standard input, output, and error devices, respectively. By exposing these members as properties, the `Console` class can delay their initialization until they are actually used. For example, upon first referencing the `Out` property, as in

```
Console.Out.WriteLine("hello, world");
```

the underlying `TextWriter` for the output device is created. But if the application makes no reference to the `In` and `Error` properties, then no objects are created for those devices.

Automatically implemented properties

An automatically implemented property (or *auto-property* for short), is a non-abstract non-extern property with semicolon-only accessor bodies. Auto-properties must have a get accessor and can optionally have a set accessor.

When a property is specified as an automatically implemented property, a hidden backing field is automatically available for the property, and the accessors are implemented to read from and write to that backing field. If the auto-property has no set accessor, the backing field is considered `readonly` ([Readonly fields](#)). Just like a `readonly` field, a getter-only auto-property can also be assigned to in the body of a constructor of the enclosing class. Such an assignment assigns directly to the readonly backing field of the property.

An auto-property may optionally have a *property_initializer*, which is applied directly to the backing field as a *variable_initializer* ([Variable initializers](#)).

The following example:


```
public class Point {
    public int X { get; set; } = 0;
    public int Y { get; set; } = 0;
}
```

is equivalent to the following declaration:

```
public class Point {
    private int __x = 0;
    private int __y = 0;
    public int X { get { return __x; } set { __x = value; } }
    public int Y { get { return __y; } set { __y = value; } }
}
```

The following example:

```
public class ReadOnlyPoint
{
    public int X { get; }
    public int Y { get; }
    public ReadOnlyPoint(int x, int y) { X = x; Y = y; }
}
```

is equivalent to the following declaration:

```
public class ReadOnlyPoint
{
    private readonly int __x;
    private readonly int __y;
    public int X { get { return __x; } }
    public int Y { get { return __y; } }
    public ReadOnlyPoint(int x, int y) { __x = x; __y = y; }
}
```

Notice that the assignments to the readonly field are legal, because they occur within the constructor.

Accessibility

If an accessor has an *accessor_modifier*, the accessibility domain ([Accessibility domains](#)) of the accessor is determined using the declared accessibility of the *accessor_modifier*. If an accessor does not have an *accessor_modifier*, the accessibility domain of the accessor is determined from the declared accessibility of the property or indexer.

The presence of an *accessor_modifier* never affects member lookup ([Operators](#)) or overload resolution ([Overload resolution](#)). The modifiers on the property or indexer always determine which property or indexer is bound to, regardless of the context of the access.

Once a particular property or indexer has been selected, the accessibility domains of the specific accessors involved are used to determine if that usage is valid:

- If the usage is as a value ([Values of expressions](#)), the `get` accessor must exist and be accessible.
- If the usage is as the target of a simple assignment ([Simple assignment](#)), the `set` accessor must exist and be accessible.
- If the usage is as the target of compound assignment ([Compound assignment](#)), or as the target of the `++` or `--` operators ([Function members.9](#), [Invocation expressions](#)), both the `get` accessors and the `set` accessor must exist and be accessible.

In the following example, the property `A.Text` is hidden by the property `B.Text`, even in contexts where only the `set` accessor is called. In contrast, the property `B.Count` is not accessible to class `M`, so the accessible property `A.Count` is used instead.

```
class A
{
    public string Text {
        get { return "hello"; }
        set { }
    }

    public int Count {
        get { return 5; }
        set { }
    }
}

class B: A
{
    private string text = "goodbye";
    private int count = 0;

    new public string Text {
        get { return text; }
        protected set { text = value; }
    }

    new protected int Count {
        get { return count; }
        set { count = value; }
    }
}

class M
{
    static void Main() {
        B b = new B();
        b.Count = 12;           // Calls A.Count set accessor
        int i = b.Count;        // Calls A.Count get accessor
        b.Text = "howdy";       // Error, B.Text set accessor not accessible
        string s = b.Text;      // Calls B.Text get accessor
    }
}
```

An accessor that is used to implement an interface may not have an *accessor_modifier*. If only one accessor is used to implement an interface, the other accessor may be declared with an *accessor_modifier*.

```
public interface I
{
    string Prop { get; }
}

public class C: I
{
    public string Prop {
        get { return "April"; }           // Must not have a modifier here
        internal set {...}                // Ok, because I.Prop has no set accessor
    }
}
```

Virtual, sealed, override, and abstract property accessors

A `virtual` property declaration specifies that the accessors of the property are virtual. The `virtual` modifier

applies to both accessors of a read-write property—it is not possible for only one accessor of a read-write property to be virtual.

An `abstract` property declaration specifies that the accessors of the property are virtual, but does not provide an actual implementation of the accessors. Instead, non-abstract derived classes are required to provide their own implementation for the accessors by overriding the property. Because an accessor for an abstract property declaration provides no actual implementation, its *accessor_body* simply consists of a semicolon.

A property declaration that includes both the `abstract` and `override` modifiers specifies that the property is abstract and overrides a base property. The accessors of such a property are also abstract.

Abstract property declarations are only permitted in abstract classes ([Abstract classes](#)). The accessors of an inherited virtual property can be overridden in a derived class by including a property declaration that specifies an `override` directive. This is known as an **overriding property declaration**. An overriding property declaration does not declare a new property. Instead, it simply specializes the implementations of the accessors of an existing virtual property.

An overriding property declaration must specify the exact same accessibility modifiers, type, and name as the inherited property. If the inherited property has only a single accessor (i.e., if the inherited property is read-only or write-only), the overriding property must include only that accessor. If the inherited property includes both accessors (i.e., if the inherited property is read-write), the overriding property can include either a single accessor or both accessors.

An overriding property declaration may include the `sealed` modifier. Use of this modifier prevents a derived class from further overriding the property. The accessors of a sealed property are also sealed.

Except for differences in declaration and invocation syntax, virtual, sealed, override, and abstract accessors behave exactly like virtual, sealed, override and abstract methods. Specifically, the rules described in [Virtual methods](#), [Override methods](#), [Sealed methods](#), and [Abstract methods](#) apply as if accessors were methods of a corresponding form:

- A `get` accessor corresponds to a parameterless method with a return value of the property type and the same modifiers as the containing property.
- A `set` accessor corresponds to a method with a single value parameter of the property type, a `void` return type, and the same modifiers as the containing property.

In the example

```
abstract class A
{
    int y;

    public virtual int X {
        get { return 0; }
    }

    public virtual int Y {
        get { return y; }
        set { y = value; }
    }

    public abstract int Z { get; set; }
}
```

`X` is a virtual read-only property, `Y` is a virtual read-write property, and `Z` is an abstract read-write property. Because `Z` is abstract, the containing class `A` must also be declared abstract.

A class that derives from `A` is shown below:

```

class B: A
{
    int z;

    public override int X {
        get { return base.X + 1; }
    }

    public override int Y {
        set { base.Y = value < 0? 0: value; }
    }

    public override int Z {
        get { return z; }
        set { z = value; }
    }
}

```

Here, the declarations of `X`, `Y`, and `Z` are overriding property declarations. Each property declaration exactly matches the accessibility modifiers, type, and name of the corresponding inherited property. The `get` accessor of `X` and the `set` accessor of `Y` use the `base` keyword to access the inherited accessors. The declaration of `Z` overrides both abstract accessors—thus, there are no outstanding abstract function members in `B`, and `B` is permitted to be a non-abstract class.

When a property is declared as an `override`, any overridden accessors must be accessible to the overriding code. In addition, the declared accessibility of both the property or indexer itself, and of the accessors, must match that of the overridden member and accessors. For example:

```

public class B
{
    public virtual int P {
        protected set {...}
        get {...}
    }
}

public class D: B
{
    public override int P {
        protected set {...}           // Must specify protected here
        get {...}                     // Must not have a modifier here
    }
}

```

Events

An **event** is a member that enables an object or class to provide notifications. Clients can attach executable code for events by supplying **event handlers**.

Events are declared using *event declarations*:

```

event_declaration
  : attributes? event_modifier* 'event' type variable_declarators ';'
  | attributes? event_modifier* 'event' type member_name '{' event_accessor_declarations '}'
  ;

event_modifier
  : 'new'
  | 'public'
  | 'protected'
  | 'internal'
  | 'private'
  | 'static'
  | 'virtual'
  | 'sealed'
  | 'override'
  | 'abstract'
  | 'extern'
  | event_modifier_unsafe
  ;

event_accessor_declarations
  : add_accessor_declaration remove_accessor_declaration
  | remove_accessor_declaration add_accessor_declaration
  ;

add_accessor_declaration
  : attributes? 'add' block
  ;

remove_accessor_declaration
  : attributes? 'remove' block
  ;

```

An *event_declaration* may include a set of *attributes* ([Attributes](#)) and a valid combination of the four access modifiers ([Access modifiers](#)), the `new` ([The new modifier](#)), `static` ([Static and instance methods](#)), `virtual` ([Virtual methods](#)), `override` ([Override methods](#)), `sealed` ([Sealed methods](#)), `abstract` ([Abstract methods](#)), and `extern` ([External methods](#)) modifiers.

Event declarations are subject to the same rules as method declarations ([Methods](#)) with regard to valid combinations of modifiers.

The *type* of an event declaration must be a *delegate_type* ([Reference types](#)), and that *delegate_type* must be at least as accessible as the event itself ([Accessibility constraints](#)).

An event declaration may include *event_accessor_declarations*. However, if it does not, for non-extern, non-abstract events, the compiler supplies them automatically ([Field-like events](#)); for extern events, the accessors are provided externally.

An event declaration that omits *event_accessor_declarations* defines one or more events—one for each of the *variable_declarators*. The attributes and modifiers apply to all of the members declared by such an *event_declaration*.

It is a compile-time error for an *event_declaration* to include both the `abstract` modifier and brace-delimited *event_accessor_declarations*.

When an event declaration includes an `extern` modifier, the event is said to be an **external event**. Because an external event declaration provides no actual implementation, it is an error for it to include both the `extern` modifier and *event_accessor_declarations*.

It is a compile-time error for a *variable_declarator* of an event declaration with an `abstract` or `external` modifier to include a *variable_initializer*.

An event can be used as the left-hand operand of the `+=` and `-=` operators ([Event assignment](#)). These operators are used, respectively, to attach event handlers to or to remove event handlers from an event, and the access modifiers of the event control the contexts in which such operations are permitted.

Since `+=` and `-=` are the only operations that are permitted on an event outside the type that declares the event, external code can add and remove handlers for an event, but cannot in any other way obtain or modify the underlying list of event handlers.

In an operation of the form `x += y` or `x -= y`, when `x` is an event and the reference takes place outside the type that contains the declaration of `x`, the result of the operation has type `void` (as opposed to having the type of `x`, with the value of `x` after the assignment). This rule prohibits external code from indirectly examining the underlying delegate of an event.

The following example shows how event handlers are attached to instances of the `Button` class:

```
public delegate void EventHandler(object sender, EventArgs e);

public class Button: Control
{
    public event EventHandler Click;
}

public class LoginDialog: Form
{
    Button OkButton;
    Button CancelButton;

    public LoginDialog() {
        OkButton = new Button(...);
        OkButton.Click += new EventHandler(OkButtonClick);
        CancelButton = new Button(...);
        CancelButton.Click += new EventHandler(CancelButtonClick);
    }

    void OkButtonClick(object sender, EventArgs e) {
        // Handle OkButton.Click event
    }

    void CancelButtonClick(object sender, EventArgs e) {
        // Handle CancelButton.Click event
    }
}
```

Here, the `LoginDialog` instance constructor creates two `Button` instances and attaches event handlers to the `Click` events.

Field-like events

Within the program text of the class or struct that contains the declaration of an event, certain events can be used like fields. To be used in this way, an event must not be `abstract` or `extern`, and must not explicitly include *event_accessor_declarations*. Such an event can be used in any context that permits a field. The field contains a delegate ([Delegates](#)) which refers to the list of event handlers that have been added to the event. If no event handlers have been added, the field contains `null`.

In the example

```

public delegate void EventHandler(object sender, EventArgs e);

public class Button: Control
{
    public event EventHandler Click;

    protected void OnClick(EventArgs e) {
        if (Click != null) Click(this, e);
    }

    public void Reset() {
        Click = null;
    }
}

```

`Click` is used as a field within the `Button` class. As the example demonstrates, the field can be examined, modified, and used in delegate invocation expressions. The `OnClick` method in the `Button` class "raises" the `Click` event. The notion of raising an event is precisely equivalent to invoking the delegate represented by the event—thus, there are no special language constructs for raising events. Note that the delegate invocation is preceded by a check that ensures the delegate is non-null.

Outside the declaration of the `Button` class, the `Click` member can only be used on the left-hand side of the `+=` and `-=` operators, as in

```
b.Click += new EventHandler(...);
```

which appends a delegate to the invocation list of the `Click` event, and

```
b.Click -= new EventHandler(...);
```

which removes a delegate from the invocation list of the `Click` event.

When compiling a field-like event, the compiler automatically creates storage to hold the delegate, and creates accessors for the event that add or remove event handlers to the delegate field. The addition and removal operations are thread safe, and may (but are not required to) be done while holding the lock ([The lock statement](#)) on the containing object for an instance event, or the type object ([Anonymous object creation expressions](#)) for a static event.

Thus, an instance event declaration of the form:

```

class X
{
    public event D Ev;
}

```

will be compiled to something equivalent to:

```

class X
{
    private D __Ev; // field to hold the delegate

    public event D Ev {
        add {
            /* add the delegate in a thread safe way */
        }

        remove {
            /* remove the delegate in a thread safe way */
        }
    }
}

```

Within the class `X`, references to `Ev` on the left-hand side of the `+=` and `-=` operators cause the add and remove accessors to be invoked. All other references to `Ev` are compiled to reference the hidden field `__Ev` instead ([Member access](#)). The name "`__Ev`" is arbitrary; the hidden field could have any name or no name at all.

Event accessors

Event declarations typically omit *event_accessor_declarations*, as in the `Button` example above. One situation for doing so involves the case in which the storage cost of one field per event is not acceptable. In such cases, a class can include *event_accessor_declarations* and use a private mechanism for storing the list of event handlers.

The *event_accessor_declarations* of an event specify the executable statements associated with adding and removing event handlers.

The accessor declarations consist of an *add_accessor_declaration* and a *remove_accessor_declaration*. Each accessor declaration consists of the token `add` or `remove` followed by a *block*. The *block* associated with an *add_accessor_declaration* specifies the statements to execute when an event handler is added, and the *block* associated with a *remove_accessor_declaration* specifies the statements to execute when an event handler is removed.

Each *add_accessor_declaration* and *remove_accessor_declaration* corresponds to a method with a single value parameter of the event type and a `void` return type. The implicit parameter of an event accessor is named `value`. When an event is used in an event assignment, the appropriate event accessor is used. Specifically, if the assignment operator is `+=` then the add accessor is used, and if the assignment operator is `-=` then the remove accessor is used. In either case, the right-hand operand of the assignment operator is used as the argument to the event accessor. The block of an *add_accessor_declaration* or a *remove_accessor_declaration* must conform to the rules for `void` methods described in [Method body](#). In particular, `return` statements in such a block are not permitted to specify an expression.

Since an event accessor implicitly has a parameter named `value`, it is a compile-time error for a local variable or constant declared in an event accessor to have that name.

In the example


```

class Control: Component
{
    // Unique keys for events
    static readonly object mouseDownEventKey = new object();
    static readonly object mouseUpEventKey = new object();

    // Return event handler associated with key
    protected Delegate GetEventHandler(object key) {...}

    // Add event handler associated with key
    protected void AddEventHandler(object key, Delegate handler) {...}

    // Remove event handler associated with key
    protected void RemoveEventHandler(object key, Delegate handler) {...}

    // MouseDown event
    public event MouseEventHandler MouseDown {
        add { AddEventHandler(mouseDownEventKey, value); }
        remove { RemoveEventHandler(mouseDownEventKey, value); }
    }

    // MouseUp event
    public event MouseEventHandler MouseUp {
        add { AddEventHandler(mouseUpEventKey, value); }
        remove { RemoveEventHandler(mouseUpEventKey, value); }
    }

    // Invoke the MouseUp event
    protected void OnMouseUp(MouseEventArgs args) {
        MouseEventHandler handler;
        handler = (MouseEventHandler)GetEventHandler(mouseUpEventKey);
        if (handler != null)
            handler(this, args);
    }
}

```

the `Control` class implements an internal storage mechanism for events. The `AddEventHandler` method associates a delegate value with a key, the `GetEventHandler` method returns the delegate currently associated with a key, and the `RemoveEventHandler` method removes a delegate as an event handler for the specified event. Presumably, the underlying storage mechanism is designed such that there is no cost for associating a `null` delegate value with a key, and thus unhandled events consume no storage.

Static and instance events

When an event declaration includes a `static` modifier, the event is said to be a *static event*. When no `static` modifier is present, the event is said to be an *instance event*.

A static event is not associated with a specific instance, and it is a compile-time error to refer to `this` in the accessors of a static event.

An instance event is associated with a given instance of a class, and this instance can be accessed as `this` ([This access](#)) in the accessors of that event.

When an event is referenced in a *member access* ([Member access](#)) of the form `E.M`, if `M` is a static event, `E` must denote a type containing `M`, and if `M` is an instance event, `E` must denote an instance of a type containing `M`.

The differences between static and instance members are discussed further in [Static and instance members](#).

Virtual, sealed, override, and abstract event accessors

A `virtual` event declaration specifies that the accessors of that event are virtual. The `virtual` modifier applies to both accessors of an event.

An `abstract` event declaration specifies that the accessors of the event are virtual, but does not provide an actual implementation of the accessors. Instead, non-abstract derived classes are required to provide their own implementation for the accessors by overriding the event. Because an abstract event declaration provides no actual implementation, it cannot provide brace-delimited *event_accessor_declarations*.

An event declaration that includes both the `abstract` and `override` modifiers specifies that the event is abstract and overrides a base event. The accessors of such an event are also abstract.

Abstract event declarations are only permitted in abstract classes ([Abstract classes](#)).

The accessors of an inherited virtual event can be overridden in a derived class by including an event declaration that specifies an `override` modifier. This is known as an ***overriding event declaration***. An overriding event declaration does not declare a new event. Instead, it simply specializes the implementations of the accessors of an existing virtual event.

An overriding event declaration must specify the exact same accessibility modifiers, type, and name as the overridden event.

An overriding event declaration may include the `sealed` modifier. Use of this modifier prevents a derived class from further overriding the event. The accessors of a sealed event are also sealed.

It is a compile-time error for an overriding event declaration to include a `new` modifier.

Except for differences in declaration and invocation syntax, virtual, sealed, override, and abstract accessors behave exactly like virtual, sealed, override and abstract methods. Specifically, the rules described in [Virtual methods](#), [Override methods](#), [Sealed methods](#), and [Abstract methods](#) apply as if accessors were methods of a corresponding form. Each accessor corresponds to a method with a single value parameter of the event type, a `void` return type, and the same modifiers as the containing event.

Indexers

An ***indexer*** is a member that enables an object to be indexed in the same way as an array. Indexers are declared using *indexer_declarations*:

```

indexer_declaration
: attributes? indexer_modifier* indexer_declarator indexer_body
;

indexer_modifier
: 'new'
| 'public'
| 'protected'
| 'internal'
| 'private'
| 'virtual'
| 'sealed'
| 'override'
| 'abstract'
| 'extern'
| indexer_modifier_unsafe
;

indexer_declarator
: type 'this' '[' formal_parameter_list '['
| type interface_type '.' 'this' '[' formal_parameter_list '['
;

indexer_body
: '{' accessor_declarations '}'
| '=>' expression ';'
;

```

An *indexer_declaration* may include a set of *attributes* ([Attributes](#)) and a valid combination of the four access modifiers ([Access modifiers](#)), the `new` ([The new modifier](#)), `virtual` ([Virtual methods](#)), `override` ([Override methods](#)), `sealed` ([Sealed methods](#)), `abstract` ([Abstract methods](#)), and `extern` ([External methods](#)) modifiers.

Indexer declarations are subject to the same rules as method declarations ([Methods](#)) with regard to valid combinations of modifiers, with the one exception being that the static modifier is not permitted on an indexer declaration.

The modifiers `virtual`, `override`, and `abstract` are mutually exclusive except in one case. The `abstract` and `override` modifiers may be used together so that an abstract indexer can override a virtual one.

The *type* of an indexer declaration specifies the element type of the indexer introduced by the declaration. Unless the indexer is an explicit interface member implementation, the *type* is followed by the keyword `this`. For an explicit interface member implementation, the *type* is followed by an *interface_type*, a `.`, and the keyword `this`. Unlike other members, indexers do not have user-defined names.

The *formal_parameter_list* specifies the parameters of the indexer. The formal parameter list of an indexer corresponds to that of a method ([Method parameters](#)), except that at least one parameter must be specified, and that the `ref` and `out` parameter modifiers are not permitted.

The *type* of an indexer and each of the types referenced in the *formal_parameter_list* must be at least as accessible as the indexer itself ([Accessibility constraints](#)).

An *indexer_body* may either consist of an *accessor body* or an *expression body*. In an accessor body, *accessor_declarations*, which must be enclosed in `{` and `}` tokens, declare the accessors ([Accessors](#)) of the property. The accessors specify the executable statements associated with reading and writing the property.

An expression body consisting of `=>` followed by an expression `E` and a semicolon is exactly equivalent to the statement body `{ get { return E; } }`, and can therefore only be used to specify getter-only indexers where the result of the getter is given by a single expression.

Even though the syntax for accessing an indexer element is the same as that for an array element, an indexer

element is not classified as a variable. Thus, it is not possible to pass an indexer element as a `ref` or `out` argument.

The formal parameter list of an indexer defines the signature ([Signatures and overloading](#)) of the indexer. Specifically, the signature of an indexer consists of the number and types of its formal parameters. The element type and names of the formal parameters are not part of an indexer's signature.

The signature of an indexer must differ from the signatures of all other indexers declared in the same class.

Indexers and properties are very similar in concept, but differ in the following ways:

- A property is identified by its name, whereas an indexer is identified by its signature.
- A property is accessed through a *simple_name* ([Simple names](#)) or a *member_access* ([Member access](#)), whereas an indexer element is accessed through an *element_access* ([Indexer access](#)).
- A property can be a `static` member, whereas an indexer is always an instance member.
- A `get` accessor of a property corresponds to a method with no parameters, whereas a `get` accessor of an indexer corresponds to a method with the same formal parameter list as the indexer.
- A `set` accessor of a property corresponds to a method with a single parameter named `value`, whereas a `set` accessor of an indexer corresponds to a method with the same formal parameter list as the indexer, plus an additional parameter named `value`.
- It is a compile-time error for an indexer accessor to declare a local variable with the same name as an indexer parameter.
- In an overriding property declaration, the inherited property is accessed using the syntax `base.P`, where `P` is the property name. In an overriding indexer declaration, the inherited indexer is accessed using the syntax `base[E]`, where `E` is a comma separated list of expressions.
- There is no concept of an "automatically implemented indexer". It is an error to have a non-abstract, non-external indexer with semicolon accessors.

Aside from these differences, all rules defined in [Accessors](#) and [Automatically implemented properties](#) apply to indexer accessors as well as to property accessors.

When an indexer declaration includes an `extern` modifier, the indexer is said to be an ***external indexer***. Because an external indexer declaration provides no actual implementation, each of its *accessor_declarations* consists of a semicolon.

The example below declares a `BitArray` class that implements an indexer for accessing the individual bits in the bit array.

```

using System;

class BitArray
{
    int[] bits;
    int length;

    public BitArray(int length) {
        if (length < 0) throw new ArgumentException();
        bits = new int[((length - 1) >> 5) + 1];
        this.length = length;
    }

    public int Length {
        get { return length; }
    }

    public bool this[int index] {
        get {
            if (index < 0 || index >= length) {
                throw new IndexOutOfRangeException();
            }
            return (bits[index >> 5] & 1 << index) != 0;
        }
        set {
            if (index < 0 || index >= length) {
                throw new IndexOutOfRangeException();
            }
            if (value) {
                bits[index >> 5] |= 1 << index;
            }
            else {
                bits[index >> 5] &= ~(1 << index);
            }
        }
    }
}

```

An instance of the `BitArray` class consumes substantially less memory than a corresponding `bool[]` (since each value of the former occupies only one bit instead of the latter's one byte), but it permits the same operations as a `bool[]`.

The following `CountPrimes` class uses a `BitArray` and the classical "sieve" algorithm to compute the number of primes between 1 and a given maximum:

```

class CountPrimes
{
    static int Count(int max) {
        BitArray flags = new BitArray(max + 1);
        int count = 1;
        for (int i = 2; i <= max; i++) {
            if (!flags[i]) {
                for (int j = i * 2; j <= max; j += i) flags[j] = true;
                count++;
            }
        }
        return count;
    }

    static void Main(string[] args) {
        int max = int.Parse(args[0]);
        int count = Count(max);
        Console.WriteLine("Found {0} primes between 1 and {1}", count, max);
    }
}

```

Note that the syntax for accessing elements of the `BitArray` is precisely the same as for a `bool[]`.

The following example shows a 26 * 10 grid class that has an indexer with two parameters. The first parameter is required to be an upper- or lowercase letter in the range A-Z, and the second is required to be an integer in the range 0-9.

```

using System;

class Grid
{
    const int NumRows = 26;
    const int NumCols = 10;

    int[, ] cells = new int[NumRows, NumCols];

    public int this[char c, int col] {
        get {
            c = Char.ToUpper(c);
            if (c < 'A' || c > 'Z') {
                throw new ArgumentException();
            }
            if (col < 0 || col >= NumCols) {
                throw new IndexOutOfRangeException();
            }
            return cells[c - 'A', col];
        }

        set {
            c = Char.ToUpper(c);
            if (c < 'A' || c > 'Z') {
                throw new ArgumentException();
            }
            if (col < 0 || col >= NumCols) {
                throw new IndexOutOfRangeException();
            }
            cells[c - 'A', col] = value;
        }
    }
}

```

Indexer overloading

The indexer overload resolution rules are described in [Type inference](#).

Operators

An ***operator*** is a member that defines the meaning of an expression operator that can be applied to instances of the class. Operators are declared using *operator_declarations*:

```
operator_declaration
: attributes? operator_modifier+ operator_declarator operator_body
;

operator_modifier
: 'public'
| 'static'
| 'extern'
| operator_modifier_unsafe
;

operator_declarator
: unary_operator_declarator
| binary_operator_declarator
| conversion_operator_declarator
;

unary_operator_declarator
: type 'operator' overloadable_unary_operator '(' type identifier ')'
;

overloadable_unary_operator
: '+' | '-' | '!' | '~' | '++' | '--' | 'true' | 'false'
;

binary_operator_declarator
: type 'operator' overloadable_binary_operator '(' type identifier ',' type identifier ')'
;

overloadable_binary_operator
: '+' | '-' | '*' | '/' | '%' | '&' | '|' | '^' | '<<'
| right_shift | '==' | '!=' | '>' | '<' | '>=' | '<='
;

conversion_operator_declarator
: 'implicit' 'operator' type '(' type identifier ')'
| 'explicit' 'operator' type '(' type identifier ')'
;

operator_body
: block
| '=>' expression ';'
| ';'
;
```

There are three categories of overloadable operators: Unary operators ([Unary operators](#)), binary operators ([Binary operators](#)), and conversion operators ([Conversion operators](#)).

The *operator_body* is either a semicolon, a ***statement body*** or an ***expression body***. A statement body consists of a *block*, which specifies the statements to execute when the operator is invoked. The *block* must conform to the rules for value-returning methods described in [Method body](#). An expression body consists of `=>` followed by an expression and a semicolon, and denotes a single expression to perform when the operator is invoked.

For `extern` operators, the *operator_body* consists simply of a semicolon. For all other operators, the *operator_body* is either a block body or an expression body.

The following rules apply to all operator declarations:

- An operator declaration must include both a `public` and a `static` modifier.
- The parameter(s) of an operator must be value parameters ([Value parameters](#)). It is a compile-time error for an operator declaration to specify `ref` or `out` parameters.
- The signature of an operator ([Unary operators](#), [Binary operators](#), [Conversion operators](#)) must differ from the signatures of all other operators declared in the same class.
- All types referenced in an operator declaration must be at least as accessible as the operator itself ([Accessibility constraints](#)).
- It is an error for the same modifier to appear multiple times in an operator declaration.

Each operator category imposes additional restrictions, as described in the following sections.

Like other members, operators declared in a base class are inherited by derived classes. Because operator declarations always require the class or struct in which the operator is declared to participate in the signature of the operator, it is not possible for an operator declared in a derived class to hide an operator declared in a base class. Thus, the `new` modifier is never required, and therefore never permitted, in an operator declaration.

Additional information on unary and binary operators can be found in [Operators](#).

Additional information on conversion operators can be found in [User-defined conversions](#).

Unary operators

The following rules apply to unary operator declarations, where `T` denotes the instance type of the class or struct that contains the operator declaration:

- A unary `+`, `-`, `!`, or `~` operator must take a single parameter of type `T` or `T?` and can return any type.
- A unary `++` or `--` operator must take a single parameter of type `T` or `T?` and must return that same type or a type derived from it.
- A unary `true` or `false` operator must take a single parameter of type `T` or `T?` and must return type `bool`.

The signature of a unary operator consists of the operator token (`+`, `-`, `!`, `~`, `++`, `--`, `true`, or `false`) and the type of the single formal parameter. The return type is not part of a unary operator's signature, nor is the name of the formal parameter.

The `true` and `false` unary operators require pair-wise declaration. A compile-time error occurs if a class declares one of these operators without also declaring the other. The `true` and `false` operators are described further in [User-defined conditional logical operators](#) and [Boolean expressions](#).

The following example shows an implementation and subsequent usage of `operator ++` for an integer vector class:


```

public class IntVector
{
    public IntVector(int length) {...}

    public int Length {...}           // read-only property

    public int this[int index] {...}   // read-write indexer

    public static IntVector operator ++(IntVector iv) {
        IntVector temp = new IntVector(iv.Length);
        for (int i = 0; i < iv.Length; i++)
            temp[i] = iv[i] + 1;
        return temp;
    }
}

class Test
{
    static void Main() {
        IntVector iv1 = new IntVector(4);    // vector of 4 x 0
        IntVector iv2;

        iv2 = iv1++;    // iv2 contains 4 x 0, iv1 contains 4 x 1
        iv2 = ++iv1;     // iv2 contains 4 x 2, iv1 contains 4 x 2
    }
}

```

Note how the operator method returns the value produced by adding 1 to the operand, just like the postfix increment and decrement operators ([Postfix increment and decrement operators](#)), and the prefix increment and decrement operators ([Prefix increment and decrement operators](#)). Unlike in C++, this method need not modify the value of its operand directly. In fact, modifying the operand value would violate the standard semantics of the postfix increment operator.

Binary operators

The following rules apply to binary operator declarations, where `T` denotes the instance type of the class or struct that contains the operator declaration:

- A binary non-shift operator must take two parameters, at least one of which must have type `T` or `T?`, and can return any type.
- A binary `<<` or `>>` operator must take two parameters, the first of which must have type `T` or `T?` and the second of which must have type `int` or `int?`, and can return any type.

The signature of a binary operator consists of the operator token (`+`, `-`, `*`, `/`, `%`, `&`, `|`, `^`, `<<`, `>>`, `==`, `!=`, `>`, `<`, `>=`, or `<=`) and the types of the two formal parameters. The return type and the names of the formal parameters are not part of a binary operator's signature.

Certain binary operators require pair-wise declaration. For every declaration of either operator of a pair, there must be a matching declaration of the other operator of the pair. Two operator declarations match when they have the same return type and the same type for each parameter. The following operators require pair-wise declaration:

- `operator ==` and `operator !=`
- `operator >` and `operator <`
- `operator >=` and `operator <=`

Conversion operators

A conversion operator declaration introduces a *user-defined conversion* ([User-defined conversions](#)) which augments the pre-defined implicit and explicit conversions.

A conversion operator declaration that includes the `implicit` keyword introduces a user-defined implicit conversion. Implicit conversions can occur in a variety of situations, including function member invocations, cast expressions, and assignments. This is described further in [Implicit conversions](#).

A conversion operator declaration that includes the `explicit` keyword introduces a user-defined explicit conversion. Explicit conversions can occur in cast expressions, and are described further in [Explicit conversions](#).

A conversion operator converts from a source type, indicated by the parameter type of the conversion operator, to a target type, indicated by the return type of the conversion operator.

For a given source type `S` and target type `T`, if `S` or `T` are nullable types, let `S0` and `T0` refer to their underlying types, otherwise `S0` and `T0` are equal to `S` and `T` respectively. A class or struct is permitted to declare a conversion from a source type `S` to a target type `T` only if all of the following are true:

- `S0` and `T0` are different types.
- Either `S0` or `T0` is the class or struct type in which the operator declaration takes place.
- Neither `S0` nor `T0` is an *interface_type*.
- Excluding user-defined conversions, a conversion does not exist from `S` to `T` or from `T` to `S`.

For the purposes of these rules, any type parameters associated with `S` or `T` are considered to be unique types that have no inheritance relationship with other types, and any constraints on those type parameters are ignored.

In the example

```
class C<T> {...}

class D<T>: C<T>
{
    public static implicit operator C<int>(D<T> value) {...}    // Ok
    public static implicit operator C<string>(D<T> value) {...} // Ok
    public static implicit operator C<T>(D<T> value) {...}     // Error
}
```

the first two operator declarations are permitted because, for the purposes of [Indexers.3](#), `T` and `int` and `string` respectively are considered unique types with no relationship. However, the third operator is an error because `C<T>` is the base class of `D<T>`.

From the second rule it follows that a conversion operator must convert either to or from the class or struct type in which the operator is declared. For example, it is possible for a class or struct type `C` to define a conversion from `C` to `int` and from `int` to `C`, but not from `int` to `bool`.

It is not possible to directly redefine a pre-defined conversion. Thus, conversion operators are not allowed to convert from or to `object` because implicit and explicit conversions already exist between `object` and all other types. Likewise, neither the source nor the target types of a conversion can be a base type of the other, since a conversion would then already exist.

However, it is possible to declare operators on generic types that, for particular type arguments, specify conversions that already exist as pre-defined conversions. In the example

```
struct Convertible<T>
{
    public static implicit operator Convertible<T>(T value) {...}
    public static explicit operator T(Convertible<T> value) {...}
}
```

when type `object` is specified as a type argument for `T`, the second operator declares a conversion that

already exists (an implicit, and therefore also an explicit, conversion exists from any type `object`).

In cases where a pre-defined conversion exists between two types, any user-defined conversions between those types are ignored. Specifically:

- If a pre-defined implicit conversion ([Implicit conversions](#)) exists from type `S` to type `T`, all user-defined conversions (implicit or explicit) from `S` to `T` are ignored.
- If a pre-defined explicit conversion ([Explicit conversions](#)) exists from type `S` to type `T`, any user-defined explicit conversions from `S` to `T` are ignored. Furthermore:

If `T` is an interface type, user-defined implicit conversions from `S` to `T` are ignored.

Otherwise, user-defined implicit conversions from `S` to `T` are still considered.

For all types but `object`, the operators declared by the `Convertible<T>` type above do not conflict with pre-defined conversions. For example:

```
void F(int i, Convertible<int> n) {
    i = n;                // Error
    i = (int)n;           // User-defined explicit conversion
    n = i;                // User-defined implicit conversion
    n = (Convertible<int>)i; // User-defined implicit conversion
}
```

However, for type `object`, pre-defined conversions hide the user-defined conversions in all cases but one:

```
void F(object o, Convertible<object> n) {
    o = n;                // Pre-defined boxing conversion
    o = (object)n;        // Pre-defined boxing conversion
    n = o;                // User-defined implicit conversion
    n = (Convertible<object>)o; // Pre-defined unboxing conversion
}
```

User-defined conversions are not allowed to convert from or to *interface_types*. In particular, this restriction ensures that no user-defined transformations occur when converting to an *interface_type*, and that a conversion to an *interface_type* succeeds only if the object being converted actually implements the specified *interface_type*.

The signature of a conversion operator consists of the source type and the target type. (Note that this is the only form of member for which the return type participates in the signature.) The `implicit` or `explicit` classification of a conversion operator is not part of the operator's signature. Thus, a class or struct cannot declare both an `implicit` and an `explicit` conversion operator with the same source and target types.

In general, user-defined implicit conversions should be designed to never throw exceptions and never lose information. If a user-defined conversion can give rise to exceptions (for example, because the source argument is out of range) or loss of information (such as discarding high-order bits), then that conversion should be defined as an explicit conversion.

In the example

```

using System;

public struct Digit
{
    byte value;

    public Digit(byte value) {
        if (value < 0 || value > 9) throw new ArgumentException();
        this.value = value;
    }

    public static implicit operator byte(Digit d) {
        return d.value;
    }

    public static explicit operator Digit(byte b) {
        return new Digit(b);
    }
}

```

the conversion from `Digit` to `byte` is implicit because it never throws exceptions or loses information, but the conversion from `byte` to `Digit` is explicit since `Digit` can only represent a subset of the possible values of a `byte`.

Instance constructors

An *instance constructor* is a member that implements the actions required to initialize an instance of a class. Instance constructors are declared using *constructor_declarations*:

```

constructor_declaration
    : attributes? constructor_modifier* constructor_declarator constructor_body
    ;

constructor_modifier
    : 'public'
    | 'protected'
    | 'internal'
    | 'private'
    | 'extern'
    | constructor_modifier_unsafe
    ;

constructor_declarator
    : identifier '(' formal_parameter_list? ')' constructor_initializer?
    ;

constructor_initializer
    : ':' 'base' '(' argument_list? ')'
    | ':' 'this' '(' argument_list? ')'
    ;

constructor_body
    : block
    | ';'
    ;

```

A *constructor_declaration* may include a set of *attributes* ([Attributes](#)), a valid combination of the four access modifiers ([Access modifiers](#)), and an `extern` ([External methods](#)) modifier. A constructor declaration is not permitted to include the same modifier multiple times.

The *identifier* of a *constructor_declarator* must name the class in which the instance constructor is declared. If

any other name is specified, a compile-time error occurs.

The optional *formal_parameter_list* of an instance constructor is subject to the same rules as the *formal_parameter_list* of a method ([Methods](#)). The formal parameter list defines the signature ([Signatures and overloading](#)) of an instance constructor and governs the process whereby overload resolution ([Type inference](#)) selects a particular instance constructor in an invocation.

Each of the types referenced in the *formal_parameter_list* of an instance constructor must be at least as accessible as the constructor itself ([Accessibility constraints](#)).

The optional *constructor_initializer* specifies another instance constructor to invoke before executing the statements given in the *constructor_body* of this instance constructor. This is described further in [Constructor initializers](#).

When a constructor declaration includes an `extern` modifier, the constructor is said to be an **external constructor**. Because an external constructor declaration provides no actual implementation, its *constructor_body* consists of a semicolon. For all other constructors, the *constructor_body* consists of a *block* which specifies the statements to initialize a new instance of the class. This corresponds exactly to the *block* of an instance method with a `void` return type ([Method body](#)).

Instance constructors are not inherited. Thus, a class has no instance constructors other than those actually declared in the class. If a class contains no instance constructor declarations, a default instance constructor is automatically provided ([Default constructors](#)).

Instance constructors are invoked by *object_creation_expressions* ([Object creation expressions](#)) and through *constructor_initializers*.

Constructor initializers

All instance constructors (except those for class `object`) implicitly include an invocation of another instance constructor immediately before the *constructor_body*. The constructor to implicitly invoke is determined by the *constructor_initializer*.

- An instance constructor initializer of the form `base(argument_list)` or `base()` causes an instance constructor from the direct base class to be invoked. That constructor is selected using *argument_list* if present and the overload resolution rules of [Overload resolution](#). The set of candidate instance constructors consists of all accessible instance constructors contained in the direct base class, or the default constructor ([Default constructors](#)), if no instance constructors are declared in the direct base class. If this set is empty, or if a single best instance constructor cannot be identified, a compile-time error occurs.
- An instance constructor initializer of the form `this(argument-list)` or `this()` causes an instance constructor from the class itself to be invoked. The constructor is selected using *argument_list* if present and the overload resolution rules of [Overload resolution](#). The set of candidate instance constructors consists of all accessible instance constructors declared in the class itself. If this set is empty, or if a single best instance constructor cannot be identified, a compile-time error occurs. If an instance constructor declaration includes a constructor initializer that invokes the constructor itself, a compile-time error occurs.

If an instance constructor has no constructor initializer, a constructor initializer of the form `base()` is implicitly provided. Thus, an instance constructor declaration of the form

```
C(...) {...}
```

is exactly equivalent to

```
C(...): base() {...}
```

The scope of the parameters given by the *formal_parameter_list* of an instance constructor declaration includes

the constructor initializer of that declaration. Thus, a constructor initializer is permitted to access the parameters of the constructor. For example:

```
class A
{
    public A(int x, int y) {}
}

class B: A
{
    public B(int x, int y): base(x + y, x - y) {}
}
```

An instance constructor initializer cannot access the instance being created. Therefore it is a compile-time error to reference `this` in an argument expression of the constructor initializer, as is it a compile-time error for an argument expression to reference any instance member through a *simple_name*.

Instance variable initializers

When an instance constructor has no constructor initializer, or it has a constructor initializer of the form `base(...)`, that constructor implicitly performs the initializations specified by the *variable_initializers* of the instance fields declared in its class. This corresponds to a sequence of assignments that are executed immediately upon entry to the constructor and before the implicit invocation of the direct base class constructor. The variable initializers are executed in the textual order in which they appear in the class declaration.

Constructor execution

Variable initializers are transformed into assignment statements, and these assignment statements are executed before the invocation of the base class instance constructor. This ordering ensures that all instance fields are initialized by their variable initializers before any statements that have access to that instance are executed.

Given the example

```
using System;

class A
{
    public A() {
        PrintFields();
    }

    public virtual void PrintFields() {}
}

class B: A
{
    int x = 1;
    int y;

    public B() {
        y = -1;
    }

    public override void PrintFields() {
        Console.WriteLine("x = {0}, y = {1}", x, y);
    }
}
```

when `new B()` is used to create an instance of `B`, the following output is produced:

```
x = 1, y = 0
```

The value of `x` is 1 because the variable initializer is executed before the base class instance constructor is invoked. However, the value of `y` is 0 (the default value of an `int`) because the assignment to `y` is not executed until after the base class constructor returns.

It is useful to think of instance variable initializers and constructor initializers as statements that are automatically inserted before the *constructor_body*. The example

```
using System;
using System.Collections;

class A
{
    int x = 1, y = -1, count;

    public A() {
        count = 0;
    }

    public A(int n) {
        count = n;
    }
}

class B: A
{
    double sqrt2 = Math.Sqrt(2.0);
    ArrayList items = new ArrayList(100);
    int max;

    public B(): this(100) {
        items.Add("default");
    }

    public B(int n): base(n - 1) {
        max = n;
    }
}
```

contains several variable initializers; it also contains constructor initializers of both forms (`base` and `this`). The example corresponds to the code shown below, where each comment indicates an automatically inserted statement (the syntax used for the automatically inserted constructor invocations isn't valid, but merely serves to illustrate the mechanism).

```

using System.Collections;

class A
{
    int x, y, count;

    public A() {
        x = 1;           // Variable initializer
        y = -1;          // Variable initializer
        object();         // Invoke object() constructor
        count = 0;
    }

    public A(int n) {
        x = 1;           // Variable initializer
        y = -1;          // Variable initializer
        object();         // Invoke object() constructor
        count = n;
    }
}

class B: A
{
    double sqrt2;
    ArrayList items;
    int max;

    public B(): this(100) {
        B(100);          // Invoke B(int) constructor
        items.Add("default");
    }

    public B(int n): base(n - 1) {
        sqrt2 = Math.Sqrt(2.0); // Variable initializer
        items = new ArrayList(100); // Variable initializer
        A(n - 1);             // Invoke A(int) constructor
        max = n;
    }
}

```

Default constructors

If a class contains no instance constructor declarations, a default instance constructor is automatically provided. That default constructor simply invokes the parameterless constructor of the direct base class. If the class is abstract then the declared accessibility for the default constructor is protected. Otherwise, the declared accessibility for the default constructor is public. Thus, the default constructor is always of the form

```
protected C(): base() {}
```

or

```
public C(): base() {}
```

where `C` is the name of the class. If overload resolution is unable to determine a unique best candidate for the base class constructor initializer then a compile-time error occurs.

In the example


```
class Message
{
    object sender;
    string text;
}
```

a default constructor is provided because the class contains no instance constructor declarations. Thus, the example is precisely equivalent to

```
class Message
{
    object sender;
    string text;

    public Message(): base() {}
}
```

Private constructors

When a class `T` declares only private instance constructors, it is not possible for classes outside the program text of `T` to derive from `T` or to directly create instances of `T`. Thus, if a class contains only static members and isn't intended to be instantiated, adding an empty private instance constructor will prevent instantiation. For example:

```
public class Trig
{
    private Trig() {}          // Prevent instantiation

    public const double PI = 3.14159265358979323846;

    public static double Sin(double x) {...}
    public static double Cos(double x) {...}
    public static double Tan(double x) {...}
}
```

The `Trig` class groups related methods and constants, but is not intended to be instantiated. Therefore it declares a single empty private instance constructor. At least one instance constructor must be declared to suppress the automatic generation of a default constructor.

Optional instance constructor parameters

The `this(...)` form of constructor initializer is commonly used in conjunction with overloading to implement optional instance constructor parameters. In the example

```
class Text
{
    public Text(): this(0, 0, null) {}

    public Text(int x, int y): this(x, y, null) {}

    public Text(int x, int y, string s) {
        // Actual constructor implementation
    }
}
```

the first two instance constructors merely provide the default values for the missing arguments. Both use a `this(...)` constructor initializer to invoke the third instance constructor, which actually does the work of initializing the new instance. The effect is that of optional constructor parameters:

```
Text t1 = new Text();           // Same as Text(0, 0, null)
Text t2 = new Text(5, 10);      // Same as Text(5, 10, null)
Text t3 = new Text(5, 20, "Hello");
```

Static constructors

A **static constructor** is a member that implements the actions required to initialize a closed class type. Static constructors are declared using *static_constructor_declarations*:

```
static_constructor_declaration
: attributes? static_constructor_modifiers identifier '(' ' ') static_constructor_body
;

static_constructor_modifiers
: 'extern'? 'static'
| 'static' 'extern'?
| static_constructor_modifiers_unsafe
;

static_constructor_body
: block
| ';'
;
```

A *static_constructor_declaration* may include a set of *attributes* ([Attributes](#)) and an `extern` modifier ([External methods](#)).

The *identifier* of a *static_constructor_declaration* must name the class in which the static constructor is declared. If any other name is specified, a compile-time error occurs.

When a static constructor declaration includes an `extern` modifier, the static constructor is said to be an **external static constructor**. Because an external static constructor declaration provides no actual implementation, its *static_constructor_body* consists of a semicolon. For all other static constructor declarations, the *static_constructor_body* consists of a *block* which specifies the statements to execute in order to initialize the class. This corresponds exactly to the *method_body* of a static method with a `void` return type ([Method body](#)).

Static constructors are not inherited, and cannot be called directly.

The static constructor for a closed class type executes at most once in a given application domain. The execution of a static constructor is triggered by the first of the following events to occur within an application domain:

- An instance of the class type is created.
- Any of the static members of the class type are referenced.

If a class contains the `Main` method ([Application Startup](#)) in which execution begins, the static constructor for that class executes before the `Main` method is called.

To initialize a new closed class type, first a new set of static fields ([Static and instance fields](#)) for that particular closed type is created. Each of the static fields is initialized to its default value ([Default values](#)). Next, the static field initializers ([Static field initialization](#)) are executed for those static fields. Finally, the static constructor is executed.

The example

```

using System;

class Test
{
    static void Main() {
        A.F();
        B.F();
    }
}

class A
{
    static A() {
        Console.WriteLine("Init A");
    }
    public static void F() {
        Console.WriteLine("A.F");
    }
}

class B
{
    static B() {
        Console.WriteLine("Init B");
    }
    public static void F() {
        Console.WriteLine("B.F");
    }
}

```

must produce the output:

```

Init A
A.F
Init B
B.F

```

because the execution of `A`'s static constructor is triggered by the call to `A.F`, and the execution of `B`'s static constructor is triggered by the call to `B.F`.

It is possible to construct circular dependencies that allow static fields with variable initializers to be observed in their default value state.

The example

```

using System;

class A
{
    public static int X;

    static A() {
        X = B.Y + 1;
    }
}

class B
{
    public static int Y = A.X + 1;

    static B() {}

    static void Main() {
        Console.WriteLine("X = {0}, Y = {1}", A.X, B.Y);
    }
}

```

produces the output

```
X = 1, Y = 2
```

To execute the `Main` method, the system first runs the initializer for `B.Y`, prior to class `B`'s static constructor. `Y`'s initializer causes `A`'s static constructor to be run because the value of `A.X` is referenced. The static constructor of `A` in turn proceeds to compute the value of `X`, and in doing so fetches the default value of `Y`, which is zero. `A.X` is thus initialized to 1. The process of running `A`'s static field initializers and static constructor then completes, returning to the calculation of the initial value of `Y`, the result of which becomes 2.

Because the static constructor is executed exactly once for each closed constructed class type, it is a convenient place to enforce run-time checks on the type parameter that cannot be checked at compile-time via constraints ([Type parameter constraints](#)). For example, the following type uses a static constructor to enforce that the type argument is an enum:

```

class Gen<T> where T: struct
{
    static Gen() {
        if (!typeof(T).IsEnum) {
            throw new ArgumentException("T must be an enum");
        }
    }
}

```

Destructors

A **destructor** is a member that implements the actions required to destruct an instance of a class. A destructor is declared using a *destructor_declaration*.

```
destructor_declaration
  : attributes? 'extern'? '~' identifier '(' ' ' ) destructor_body
  | destructor_declaration_unsafe
  ;

destructor_body
  : block
  | ';'
  ;
```

A *destructor_declaration* may include a set of *attributes* ([Attributes](#)).

The *identifier* of a *destructor_declaration* must name the class in which the destructor is declared. If any other name is specified, a compile-time error occurs.

When a destructor declaration includes an `extern` modifier, the destructor is said to be an ***external destructor***. Because an external destructor declaration provides no actual implementation, its *destructor_body* consists of a semicolon. For all other destructors, the *destructor_body* consists of a *block* which specifies the statements to execute in order to destruct an instance of the class. A *destructor_body* corresponds exactly to the *method_body* of an instance method with a `void` return type ([Method body](#)).

Destructors are not inherited. Thus, a class has no destructors other than the one which may be declared in that class.

Since a destructor is required to have no parameters, it cannot be overloaded, so a class can have, at most, one destructor.

Destructors are invoked automatically, and cannot be invoked explicitly. An instance becomes eligible for destruction when it is no longer possible for any code to use that instance. Execution of the destructor for the instance may occur at any time after the instance becomes eligible for destruction. When an instance is destructed, the destructors in that instance's inheritance chain are called, in order, from most derived to least derived. A destructor may be executed on any thread. For further discussion of the rules that govern when and how a destructor is executed, see [Automatic memory management](#).

The output of the example

```

using System;

class A
{
    ~A() {
        Console.WriteLine("A's destructor");
    }
}

class B: A
{
    ~B() {
        Console.WriteLine("B's destructor");
    }
}

class Test
{
    static void Main() {
        B b = new B();
        b = null;
        GC.Collect();
        GC.WaitForPendingFinalizers();
    }
}

```

is

```

B's destructor
A's destructor

```

since destructors in an inheritance chain are called in order, from most derived to least derived.

Destructors are implemented by overriding the virtual method `Finalize` on `System.Object`. C# programs are not permitted to override this method or call it (or overrides of it) directly. For instance, the program

```

class A
{
    override protected void Finalize() {}    // error

    public void F() {
        this.Finalize();                      // error
    }
}

```

contains two errors.

The compiler behaves as if this method, and overrides of it, do not exist at all. Thus, this program:

```

class A
{
    void Finalize() {}                        // permitted
}

```

is valid, and the method shown hides `System.Object`'s `Finalize` method.

For a discussion of the behavior when an exception is thrown from a destructor, see [How exceptions are handled](#).

Iterators

A function member ([Function members](#)) implemented using an iterator block ([Blocks](#)) is called an *iterator*.

An iterator block may be used as the body of a function member as long as the return type of the corresponding function member is one of the enumerator interfaces ([Enumerator interfaces](#)) or one of the enumerable interfaces ([Enumerable interfaces](#)). It can occur as a *method_body*, *operator_body* or *accessor_body*, whereas events, instance constructors, static constructors and destructors cannot be implemented as iterators.

When a function member is implemented using an iterator block, it is a compile-time error for the formal parameter list of the function member to specify any `ref` or `out` parameters.

Enumerator interfaces

The *enumerator interfaces* are the non-generic interface `System.Collections.IEnumerator` and all instantiations of the generic interface `System.Collections.Generic.IEnumerator<T>`. For the sake of brevity, in this chapter these interfaces are referenced as `IEnumerator` and `IEnumerator<T>`, respectively.

Enumerable interfaces

The *enumerable interfaces* are the non-generic interface `System.Collections.IEnumerable` and all instantiations of the generic interface `System.Collections.Generic.IEnumerable<T>`. For the sake of brevity, in this chapter these interfaces are referenced as `IEnumerable` and `IEnumerable<T>`, respectively.

Yield type

An iterator produces a sequence of values, all of the same type. This type is called the *yield type* of the iterator.

- The yield type of an iterator that returns `IEnumerator` or `IEnumerable` is `object`.
- The yield type of an iterator that returns `IEnumerator<T>` or `IEnumerable<T>` is `T`.

Enumerator objects

When a function member returning an enumerator interface type is implemented using an iterator block, invoking the function member does not immediately execute the code in the iterator block. Instead, an *enumerator object* is created and returned. This object encapsulates the code specified in the iterator block, and execution of the code in the iterator block occurs when the enumerator object's `MoveNext` method is invoked. An enumerator object has the following characteristics:

- It implements `IEnumerator` and `IEnumerator<T>`, where `T` is the yield type of the iterator.
- It implements `System.IDisposable`.
- It is initialized with a copy of the argument values (if any) and instance value passed to the function member.
- It has four potential states, *before*, *running*, *suspended*, and *after*, and is initially in the *before* state.

An enumerator object is typically an instance of a compiler-generated enumerator class that encapsulates the code in the iterator block and implements the enumerator interfaces, but other methods of implementation are possible. If an enumerator class is generated by the compiler, that class will be nested, directly or indirectly, in the class containing the function member, it will have private accessibility, and it will have a name reserved for compiler use ([Identifiers](#)).

An enumerator object may implement more interfaces than those specified above.

The following sections describe the exact behavior of the `MoveNext`, `Current`, and `Dispose` members of the `IEnumerable` and `IEnumerable<T>` interface implementations provided by an enumerator object.

Note that enumerator objects do not support the `IEnumerator.Reset` method. Invoking this method causes a `System.NotSupportedException` to be thrown.

The MoveNext method

The `MoveNext` method of an enumerator object encapsulates the code of an iterator block. Invoking the

`MoveNext` method executes code in the iterator block and sets the `Current` property of the enumerator object as appropriate. The precise action performed by `MoveNext` depends on the state of the enumerator object when `MoveNext` is invoked:

- If the state of the enumerator object is *before*, invoking `MoveNext` :
 - Changes the state to *running*.
 - Initializes the parameters (including `this`) of the iterator block to the argument values and instance value saved when the enumerator object was initialized.
 - Executes the iterator block from the beginning until execution is interrupted (as described below).
- If the state of the enumerator object is *running*, the result of invoking `MoveNext` is unspecified.
- If the state of the enumerator object is *suspended*, invoking `MoveNext` :
 - Changes the state to *running*.
 - Restores the values of all local variables and parameters (including `this`) to the values saved when execution of the iterator block was last suspended. Note that the contents of any objects referenced by these variables may have changed since the previous call to `MoveNext`.
 - Resumes execution of the iterator block immediately following the `yield return` statement that caused the suspension of execution and continues until execution is interrupted (as described below).
- If the state of the enumerator object is *after*, invoking `MoveNext` returns `false`.

When `MoveNext` executes the iterator block, execution can be interrupted in four ways: By a `yield return` statement, by a `yield break` statement, by encountering the end of the iterator block, and by an exception being thrown and propagated out of the iterator block.

- When a `yield return` statement is encountered ([The yield statement](#)):
 - The expression given in the statement is evaluated, implicitly converted to the `yield` type, and assigned to the `Current` property of the enumerator object.
 - Execution of the iterator body is suspended. The values of all local variables and parameters (including `this`) are saved, as is the location of this `yield return` statement. If the `yield return` statement is within one or more `try` blocks, the associated `finally` blocks are not executed at this time.
 - The state of the enumerator object is changed to *suspended*.
 - The `MoveNext` method returns `true` to its caller, indicating that the iteration successfully advanced to the next value.
- When a `yield break` statement is encountered ([The yield statement](#)):
 - If the `yield break` statement is within one or more `try` blocks, the associated `finally` blocks are executed.
 - The state of the enumerator object is changed to *after*.
 - The `MoveNext` method returns `false` to its caller, indicating that the iteration is complete.
- When the end of the iterator body is encountered:
 - The state of the enumerator object is changed to *after*.
 - The `MoveNext` method returns `false` to its caller, indicating that the iteration is complete.
- When an exception is thrown and propagated out of the iterator block:
 - Appropriate `finally` blocks in the iterator body will have been executed by the exception propagation.
 - The state of the enumerator object is changed to *after*.
 - The exception propagation continues to the caller of the `MoveNext` method.

The Current property

An enumerator object's `Current` property is affected by `yield return` statements in the iterator block.

When an enumerator object is in the *suspended* state, the value of `Current` is the value set by the previous call

to `MoveNext`. When an enumerator object is in the *before*, *running*, or *after* states, the result of accessing `Current` is unspecified.

For an iterator with a yield type other than `object`, the result of accessing `Current` through the enumerator object's `IEnumerable` implementation corresponds to accessing `Current` through the enumerator object's `IEnumerator<T>` implementation and casting the result to `object`.

The Dispose method

The `Dispose` method is used to clean up the iteration by bringing the enumerator object to the *after* state.

- If the state of the enumerator object is *before*, invoking `Dispose` changes the state to *after*.
- If the state of the enumerator object is *running*, the result of invoking `Dispose` is unspecified.
- If the state of the enumerator object is *suspended*, invoking `Dispose`:
 - Changes the state to *running*.
 - Executes any finally blocks as if the last executed `yield return` statement were a `yield break` statement. If this causes an exception to be thrown and propagated out of the iterator body, the state of the enumerator object is set to *after* and the exception is propagated to the caller of the `Dispose` method.
 - Changes the state to *after*.
- If the state of the enumerator object is *after*, invoking `Dispose` has no effect.

Enumerable objects

When a function member returning an enumerable interface type is implemented using an iterator block, invoking the function member does not immediately execute the code in the iterator block. Instead, an **enumerable object** is created and returned. The enumerable object's `GetEnumerator` method returns an enumerator object that encapsulates the code specified in the iterator block, and execution of the code in the iterator block occurs when the enumerator object's `MoveNext` method is invoked. An enumerable object has the following characteristics:

- It implements `IEnumerable` and `IEnumerator<T>`, where `T` is the yield type of the iterator.
- It is initialized with a copy of the argument values (if any) and instance value passed to the function member.

An enumerable object is typically an instance of a compiler-generated enumerable class that encapsulates the code in the iterator block and implements the enumerable interfaces, but other methods of implementation are possible. If an enumerable class is generated by the compiler, that class will be nested, directly or indirectly, in the class containing the function member, it will have private accessibility, and it will have a name reserved for compiler use ([Identifiers](#)).

An enumerable object may implement more interfaces than those specified above. In particular, an enumerable object may also implement `IEnumerator` and `IEnumerator<T>`, enabling it to serve as both an enumerable and an enumerator. In that type of implementation, the first time an enumerable object's `GetEnumerator` method is invoked, the enumerable object itself is returned. Subsequent invocations of the enumerable object's `GetEnumerator`, if any, return a copy of the enumerable object. Thus, each returned enumerator has its own state and changes in one enumerator will not affect another.

The GetEnumerator method

An enumerable object provides an implementation of the `GetEnumerator` methods of the `IEnumerable` and `IEnumerator<T>` interfaces. The two `GetEnumerator` methods share a common implementation that acquires and returns an available enumerator object. The enumerator object is initialized with the argument values and instance value saved when the enumerable object was initialized, but otherwise the enumerator object functions as described in [Enumerator objects](#).

Implementation example

This section describes a possible implementation of iterators in terms of standard C# constructs. The

implementation described here is based on the same principles used by the Microsoft C# compiler, but it is by no means a mandated implementation or the only one possible.

The following `Stack<T>` class implements its `GetEnumerator` method using an iterator. The iterator enumerates the elements of the stack in top to bottom order.

```
using System;
using System.Collections;
using System.Collections.Generic;

class Stack<T>: IEnumerable<T>
{
    T[] items;
    int count;

    public void Push(T item) {
        if (items == null) {
            items = new T[4];
        }
        else if (items.Length == count) {
            T[] newItems = new T[count * 2];
            Array.Copy(items, 0, newItems, 0, count);
            items = newItems;
        }
        items[count++] = item;
    }

    public T Pop() {
        T result = items[--count];
        items[count] = default(T);
        return result;
    }

    public IEnumerator<T> GetEnumerator() {
        for (int i = count - 1; i >= 0; --i) yield return items[i];
    }
}
```

The `GetEnumerator` method can be translated into an instantiation of a compiler-generated enumerator class that encapsulates the code in the iterator block, as shown in the following.

```

class Stack<T>: IEnumerable<T>
{
    ...

    public IEnumerator<T> GetEnumerator() {
        return new __Enumerator1(this);
    }

    class __Enumerator1: IEnumerator<T>, IEnumerator
    {
        int __state;
        T __current;
        Stack<T> __this;
        int i;

        public __Enumerator1(Stack<T> __this) {
            this.__this = __this;
        }

        public T Current {
            get { return __current; }
        }

        object IEnumerator.Current {
            get { return __current; }
        }

        public bool MoveNext() {
            switch (__state) {
                case 1: goto __state1;
                case 2: goto __state2;
            }
            i = __this.count - 1;
            __loop:
            if (i < 0) goto __state2;
            __current = __this.items[i];
            __state = 1;
            return true;
            __state1:
            --i;
            goto __loop;
            __state2:
            __state = 2;
            return false;
        }

        public void Dispose() {
            __state = 2;
        }

        void IEnumerator.Reset() {
            throw new NotSupportedException();
        }
    }
}

```

In the preceding translation, the code in the iterator block is turned into a state machine and placed in the `MoveNext` method of the enumerator class. Furthermore, the local variable `i` is turned into a field in the enumerator object so it can continue to exist across invocations of `MoveNext`.

The following example prints a simple multiplication table of the integers 1 through 10. The `FromTo` method in the example returns an enumerable object and is implemented using an iterator.

```

using System;
using System.Collections.Generic;

class Test
{
    static IEnumerable<int> FromTo(int from, int to) {
        while (from <= to) yield return from++;
    }

    static void Main() {
        IEnumerable<int> e = FromTo(1, 10);
        foreach (int x in e) {
            foreach (int y in e) {
                Console.Write("{0,3} ", x * y);
            }
            Console.WriteLine();
        }
    }
}

```

The `FromTo` method can be translated into an instantiation of a compiler-generated enumerable class that encapsulates the code in the iterator block, as shown in the following.

```

using System;
using System.Threading;
using System.Collections;
using System.Collections.Generic;

class Test
{
    ...

    static IEnumerable<int> FromTo(int from, int to) {
        return new __Enumerable1(from, to);
    }

    class __Enumerable1:
        IEnumerable<int>, IEnumerable,
        IEnumerator<int>, IEnumerator
    {
        int __state;
        int __current;
        int __from;
        int from;
        int to;
        int i;

        public __Enumerable1(int __from, int to) {
            this.__from = __from;
            this.to = to;
        }

        public IEnumerator<int> GetEnumerator() {
            __Enumerable1 result = this;
            if (Interlocked.CompareExchange(ref __state, 1, 0) != 0) {
                result = new __Enumerable1(__from, to);
                result.__state = 1;
            }
            result.from = result.__from;
            return result;
        }

        IEnumerator IEnumerable.GetEnumerator() {
            return (IEnumerator)GetEnumerator();
        }
    }
}

```

```

    public int Current {
        get { return __current; }
    }

    object IEnumerator.Current {
        get { return __current; }
    }

    public bool MoveNext() {
        switch (__state) {
            case 1:
                if (from > to) goto case 2;
                __current = from++;
                __state = 1;
                return true;
            case 2:
                __state = 2;
                return false;
            default:
                throw new InvalidOperationException();
        }
    }

    public void Dispose() {
        __state = 2;
    }

    void IEnumerator.Reset() {
        throw new NotSupportedException();
    }
}

```

The enumerable class implements both the enumerable interfaces and the enumerator interfaces, enabling it to serve as both an enumerable and an enumerator. The first time the `GetEnumerator` method is invoked, the enumerable object itself is returned. Subsequent invocations of the enumerable object's `GetEnumerator`, if any, return a copy of the enumerable object. Thus, each returned enumerator has its own state and changes in one enumerator will not affect another. The `Interlocked.CompareExchange` method is used to ensure thread-safe operation.

The `from` and `to` parameters are turned into fields in the enumerable class. Because `from` is modified in the iterator block, an additional `__from` field is introduced to hold the initial value given to `from` in each enumerator.

The `MoveNext` method throws an `InvalidOperationException` if it is called when `__state` is `0`. This protects against use of the enumerable object as an enumerator object without first calling `GetEnumerator`.

The following example shows a simple tree class. The `Tree<T>` class implements its `GetEnumerator` method using an iterator. The iterator enumerates the elements of the tree in infix order.

```

using System;
using System.Collections.Generic;

class Tree<T>: IEnumerable<T>
{
    T value;
    Tree<T> left;
    Tree<T> right;

    public Tree(T value, Tree<T> left, Tree<T> right) {
        this.value = value;
        this.left = left;
        this.right = right;
    }

    public IEnumerator<T> GetEnumerator() {
        if (left != null) foreach (T x in left) yield x;
        yield value;
        if (right != null) foreach (T x in right) yield x;
    }
}

class Program
{
    static Tree<T> MakeTree<T>(T[] items, int left, int right) {
        if (left > right) return null;
        int i = (left + right) / 2;
        return new Tree<T>(items[i],
            MakeTree(items, left, i - 1),
            MakeTree(items, i + 1, right));
    }

    static Tree<T> MakeTree<T>(params T[] items) {
        return MakeTree(items, 0, items.Length - 1);
    }

    // The output of the program is:
    // 1 2 3 4 5 6 7 8 9
    // Mon Tue Wed Thu Fri Sat Sun

    static void Main() {
        Tree<int> ints = MakeTree(1, 2, 3, 4, 5, 6, 7, 8, 9);
        foreach (int i in ints) Console.Write("{0} ", i);
        Console.WriteLine();

        Tree<string> strings = MakeTree(
            "Mon", "Tue", "Wed", "Thu", "Fri", "Sat", "Sun");
        foreach (string s in strings) Console.Write("{0} ", s);
        Console.WriteLine();
    }
}

```

The `GetEnumerator` method can be translated into an instantiation of a compiler-generated enumerator class that encapsulates the code in the iterator block, as shown in the following.

```

class Tree<T>: IEnumerable<T>
{
    ...

    public IEnumerator<T> GetEnumerator() {
        return new __Enumerator1(this);
    }

    class __Enumerator1 : IEnumerator<T>, IEnumerator
    {
        Node<T> this:

```

```

Node<T> __this;
IEnumerator<T> __left, __right;
int __state;
T __current;

public __Enumerator1(Node<T> __this) {
    this.__this = __this;
}

public T Current {
    get { return __current; }
}

object IEnumerator.Current {
    get { return __current; }
}

public bool MoveNext() {
    try {
        switch (__state) {

            case 0:
                __state = -1;
                if (__this.left == null) goto __yield_value;
                __left = __this.left.GetEnumerator();
                goto case 1;

            case 1:
                __state = -2;
                if (!__left.MoveNext()) goto __left_dispose;
                __current = __left.Current;
                __state = 1;
                return true;

            __left_dispose:
                __state = -1;
                __left.Dispose();

            __yield_value:
                __current = __this.value;
                __state = 2;
                return true;

            case 2:
                __state = -1;
                if (__this.right == null) goto __end;
                __right = __this.right.GetEnumerator();
                goto case 3;

            case 3:
                __state = -3;
                if (!__right.MoveNext()) goto __right_dispose;
                __current = __right.Current;
                __state = 3;
                return true;

            __right_dispose:
                __state = -1;
                __right.Dispose();

            __end:
                __state = 4;
                break;

        }
    }
    finally {
        if (__state < 0) Dispose();
    }
    return false;
}

```

```

        return false;
    }

    public void Dispose() {
        try {
            switch (__state) {

                case 1:
                case -2:
                    __left.Dispose();
                    break;

                case 3:
                case -3:
                    __right.Dispose();
                    break;

            }
        }
        finally {
            __state = 4;
        }
    }

    void IEnumerator.Reset() {
        throw new NotSupportedException();
    }
}

```

The compiler generated temporaries used in the `foreach` statements are lifted into the `__left` and `__right` fields of the enumerator object. The `__state` field of the enumerator object is carefully updated so that the correct `Dispose()` method will be called correctly if an exception is thrown. Note that it is not possible to write the translated code with simple `foreach` statements.

Async functions

A method ([Methods](#)) or anonymous function ([Anonymous function expressions](#)) with the `async` modifier is called an *async function*. In general, the term *async* is used to describe any kind of function that has the `async` modifier.

It is a compile-time error for the formal parameter list of an async function to specify any `ref` or `out` parameters.

The *return_type* of an async method must be either `void` or a *task type*. The task types are `System.Threading.Tasks.Task` and types constructed from `System.Threading.Tasks.Task<T>`. For the sake of brevity, in this chapter these types are referenced as `Task` and `Task<T>`, respectively. An async method returning a task type is said to be task-returning.

The exact definition of the task types is implementation defined, but from the language's point of view a task type is in one of the states incomplete, succeeded or faulted. A faulted task records a pertinent exception. A succeeded `Task<T>` records a result of type `T`. Task types are awaitable, and can therefore be the operands of await expressions ([Await expressions](#)).

An async function invocation has the ability to suspend evaluation by means of await expressions ([Await expressions](#)) in its body. Evaluation may later be resumed at the point of the suspending await expression by means of a *resumption delegate*. The resumption delegate is of type `System.Action`, and when it is invoked, evaluation of the async function invocation will resume from the await expression where it left off. The *current caller* of an async function invocation is the original caller if the function invocation has never been suspended, or the most recent caller of the resumption delegate otherwise.

Evaluation of a task-returning async function

Invocation of a task-returning async function causes an instance of the returned task type to be generated. This is called the *return task* of the async function. The task is initially in an incomplete state.

The async function body is then evaluated until it is either suspended (by reaching an await expression) or terminates, at which point control is returned to the caller, along with the return task.

When the body of the async function terminates, the return task is moved out of the incomplete state:

- If the function body terminates as the result of reaching a return statement or the end of the body, any result value is recorded in the return task, which is put into a succeeded state.
- If the function body terminates as the result of an uncaught exception ([The throw statement](#)) the exception is recorded in the return task which is put into a faulted state.

Evaluation of a void-returning async function

If the return type of the async function is `void`, evaluation differs from the above in the following way: Because no task is returned, the function instead communicates completion and exceptions to the current thread's *synchronization context*. The exact definition of synchronization context is implementation-dependent, but is a representation of "where" the current thread is running. The synchronization context is notified when evaluation of a void-returning async function commences, completes successfully, or causes an uncaught exception to be thrown.

This allows the context to keep track of how many void-returning async functions are running under it, and to decide how to propagate exceptions coming out of them.

Structs

12/28/2021 • 19 minutes to read • [Edit Online](#)

Structs are similar to classes in that they represent data structures that can contain data members and function members. However, unlike classes, structs are value types and do not require heap allocation. A variable of a struct type directly contains the data of the struct, whereas a variable of a class type contains a reference to the data, the latter known as an object.

Structs are particularly useful for small data structures that have value semantics. Complex numbers, points in a coordinate system, or key-value pairs in a dictionary are all good examples of structs. Key to these data structures is that they have few data members, that they do not require use of inheritance or referential identity, and that they can be conveniently implemented using value semantics where assignment copies the value instead of the reference.

As described in [Simple types](#), the simple types provided by C#, such as `int`, `double`, and `bool`, are in fact all struct types. Just as these predefined types are structs, it is also possible to use structs and operator overloading to implement new "primitive" types in the C# language. Two examples of such types are given at the end of this chapter ([Struct examples](#)).

Struct declarations

A *struct_declaration* is a *type_declaration* ([Type declarations](#)) that declares a new struct:

```
struct_declaration
: attributes? struct_modifier* 'partial'? 'struct' identifier type_parameter_list?
  struct_interfaces? type_parameter_constraints_clause* struct_body ';'?
```

A *struct_declaration* consists of an optional set of *attributes* ([Attributes](#)), followed by an optional set of *struct_modifiers* ([Struct modifiers](#)), followed by an optional `partial` modifier, followed by the keyword `struct` and an *identifier* that names the struct, followed by an optional *type_parameter_list* specification ([Type parameters](#)), followed by an optional *struct_interfaces* specification ([Partial modifier](#)), followed by an optional *type_parameter_constraints_clauses* specification ([Type parameter constraints](#)), followed by a *struct_body* ([Struct body](#)), optionally followed by a semicolon.

Struct modifiers

A *struct_declaration* may optionally include a sequence of struct modifiers:

```
struct_modifier
: 'new'
| 'public'
| 'protected'
| 'internal'
| 'private'
| struct_modifier_unsafe
;
```

It is a compile-time error for the same modifier to appear multiple times in a struct declaration.

The modifiers of a struct declaration have the same meaning as those of a class declaration ([Class declarations](#)).

Partial modifier

The `partial` modifier indicates that this *struct_declaration* is a partial type declaration. Multiple partial struct declarations with the same name within an enclosing namespace or type declaration combine to form one struct declaration, following the rules specified in [Partial types](#).

Struct interfaces

A struct declaration may include a *struct_interfaces* specification, in which case the struct is said to directly implement the given interface types.

```
struct_interfaces
: ':' interface_type_list
;
```

Interface implementations are discussed further in [Interface implementations](#).

Struct body

The *struct_body* of a struct defines the members of the struct.

```
struct_body
: '{' struct_member_declaration* '}'
;
```

Struct members

The members of a struct consist of the members introduced by its *struct_member_declarations* and the members inherited from the type `System.ValueType`.

```
struct_member_declaration
: constant_declaration
| field_declaration
| method_declaration
| property_declaration
| event_declaration
| indexer_declaration
| operator_declaration
| constructor_declaration
| static_constructor_declaration
| type_declaration
| struct_member_declaration_unsafe
;
```

Except for the differences noted in [Class and struct differences](#), the descriptions of class members provided in [Class members](#) through [Async functions](#) apply to struct members as well.

Class and struct differences

Structs differ from classes in several important ways:

- Structs are value types ([Value semantics](#)).
- All struct types implicitly inherit from the class `System.ValueType` ([Inheritance](#)).
- Assignment to a variable of a struct type creates a copy of the value being assigned ([Assignment](#)).
- The default value of a struct is the value produced by setting all value type fields to their default value and all reference type fields to `null` ([Default values](#)).
- Boxing and unboxing operations are used to convert between a struct type and `object` ([Boxing and unboxing](#)).

- The meaning of `this` is different for structs ([This access](#)).
- Instance field declarations for a struct are not permitted to include variable initializers ([Field initializers](#)).
- A struct is not permitted to declare a parameterless instance constructor ([Constructors](#)).
- A struct is not permitted to declare a destructor ([Destructors](#)).

Value semantics

Structs are value types ([Value types](#)) and are said to have value semantics. Classes, on the other hand, are reference types ([Reference types](#)) and are said to have reference semantics.

A variable of a struct type directly contains the data of the struct, whereas a variable of a class type contains a reference to the data, the latter known as an object. When a struct `B` contains an instance field of type `A` and `A` is a struct type, it is a compile-time error for `A` to depend on `B` or a type constructed from `B`. A struct `x` ***directly depends on*** a struct `y` if `x` contains an instance field of type `y`. Given this definition, the complete set of structs upon which a struct depends is the transitive closure of the ***directly depends on*** relationship. For example

```
struct Node
{
    int data;
    Node next; // error, Node directly depends on itself
}
```

is an error because `Node` contains an instance field of its own type. Another example

```
struct A { B b; }

struct B { C c; }

struct C { A a; }
```

is an error because each of the types `A`, `B`, and `C` depend on each other.

With classes, it is possible for two variables to reference the same object, and thus possible for operations on one variable to affect the object referenced by the other variable. With structs, the variables each have their own copy of the data (except in the case of `ref` and `out` parameter variables), and it is not possible for operations on one to affect the other. Furthermore, because structs are not reference types, it is not possible for values of a struct type to be `null`.

Given the declaration

```
struct Point
{
    public int x, y;

    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }
}
```

the code fragment

```
Point a = new Point(10, 10);
Point b = a;
a.x = 100;
System.Console.WriteLine(b.x);
```

outputs the value `10`. The assignment of `a` to `b` creates a copy of the value, and `b` is thus unaffected by the assignment to `a.x`. Had `Point` instead been declared as a class, the output would be `100` because `a` and `b` would reference the same object.

Inheritance

All struct types implicitly inherit from the class `System.ValueType`, which, in turn, inherits from class `object`. A struct declaration may specify a list of implemented interfaces, but it is not possible for a struct declaration to specify a base class.

Struct types are never abstract and are always implicitly sealed. The `abstract` and `sealed` modifiers are therefore not permitted in a struct declaration.

Since inheritance isn't supported for structs, the declared accessibility of a struct member cannot be `protected` or `protected internal`.

Function members in a struct cannot be `abstract` or `virtual`, and the `override` modifier is allowed only to override methods inherited from `System.ValueType`.

Assignment

Assignment to a variable of a struct type creates a copy of the value being assigned. This differs from assignment to a variable of a class type, which copies the reference but not the object identified by the reference.

Similar to an assignment, when a struct is passed as a value parameter or returned as the result of a function member, a copy of the struct is created. A struct may be passed by reference to a function member using a `ref` or `out` parameter.

When a property or indexer of a struct is the target of an assignment, the instance expression associated with the property or indexer access must be classified as a variable. If the instance expression is classified as a value, a compile-time error occurs. This is described in further detail in [Simple assignment](#).

Default values

As described in [Default values](#), several kinds of variables are automatically initialized to their default value when they are created. For variables of class types and other reference types, this default value is `null`. However, since structs are value types that cannot be `null`, the default value of a struct is the value produced by setting all value type fields to their default value and all reference type fields to `null`.

Referring to the `Point` struct declared above, the example

```
Point[] a = new Point[100];
```

initializes each `Point` in the array to the value produced by setting the `x` and `y` fields to zero.

The default value of a struct corresponds to the value returned by the default constructor of the struct ([Default constructors](#)). Unlike a class, a struct is not permitted to declare a parameterless instance constructor. Instead, every struct implicitly has a parameterless instance constructor which always returns the value that results from setting all value type fields to their default value and all reference type fields to `null`.

Structs should be designed to consider the default initialization state a valid state. In the example

```

using System;

struct KeyValuePair
{
    string key;
    string value;

    public KeyValuePair(string key, string value) {
        if (key == null || value == null) throw new ArgumentException();
        this.key = key;
        this.value = value;
    }
}

```

the user-defined instance constructor protects against null values only where it is explicitly called. In cases where a `KeyValuePair` variable is subject to default value initialization, the `key` and `value` fields will be null, and the struct must be prepared to handle this state.

Boxing and unboxing

A value of a class type can be converted to type `object` or to an interface type that is implemented by the class simply by treating the reference as another type at compile-time. Likewise, a value of type `object` or a value of an interface type can be converted back to a class type without changing the reference (but of course a run-time type check is required in this case).

Since structs are not reference types, these operations are implemented differently for struct types. When a value of a struct type is converted to type `object` or to an interface type that is implemented by the struct, a boxing operation takes place. Likewise, when a value of type `object` or a value of an interface type is converted back to a struct type, an unboxing operation takes place. A key difference from the same operations on class types is that boxing and unboxing copies the struct value either into or out of the boxed instance. Thus, following a boxing or unboxing operation, changes made to the unboxed struct are not reflected in the boxed struct.

When a struct type overrides a virtual method inherited from `System.Object` (such as `Equals`, `GetHashCode`, or `ToString`), invocation of the virtual method through an instance of the struct type does not cause boxing to occur. This is true even when the struct is used as a type parameter and the invocation occurs through an instance of the type parameter type. For example:

```

using System;

struct Counter
{
    int value;

    public override string ToString() {
        value++;
        return value.ToString();
    }
}

class Program
{
    static void Test<T>() where T: new() {
        T x = new T();
        Console.WriteLine(x.ToString());
        Console.WriteLine(x.ToString());
        Console.WriteLine(x.ToString());
    }

    static void Main() {
        Test<Counter>();
    }
}

```

The output of the program is:

```

1
2
3

```

Although it is bad style for `ToString` to have side effects, the example demonstrates that no boxing occurred for the three invocations of `x.ToString()`.

Similarly, boxing never implicitly occurs when accessing a member on a constrained type parameter. For example, suppose an interface `ICounter` contains a method `Increment` which can be used to modify a value. If `ICounter` is used as a constraint, the implementation of the `Increment` method is called with a reference to the variable that `Increment` was called on, never a boxed copy.

```

using System;

interface ICounter
{
    void Increment();
}

struct Counter: ICounter
{
    int value;

    public override string ToString() {
        return value.ToString();
    }

    void ICounter.Increment() {
        value++;
    }
}

class Program
{
    static void Test<T>() where T: ICounter, new() {
        T x = new T();
        Console.WriteLine(x);
        x.Increment();           // Modify x
        Console.WriteLine(x);
        ((ICounter)x).Increment(); // Modify boxed copy of x
        Console.WriteLine(x);
    }

    static void Main() {
        Test<Counter>();
    }
}

```

The first call to `Increment` modifies the value in the variable `x`. This is not equivalent to the second call to `Increment`, which modifies the value in a boxed copy of `x`. Thus, the output of the program is:

```

0
1
1

```

For further details on boxing and unboxing, see [Boxing and unboxing](#).

Meaning of this

Within an instance constructor or instance function member of a class, `this` is classified as a value. Thus, while `this` can be used to refer to the instance for which the function member was invoked, it is not possible to assign to `this` in a function member of a class.

Within an instance constructor of a struct, `this` corresponds to an `out` parameter of the struct type, and within an instance function member of a struct, `this` corresponds to a `ref` parameter of the struct type. In both cases, `this` is classified as a variable, and it is possible to modify the entire struct for which the function member was invoked by assigning to `this` or by passing this as a `ref` or `out` parameter.

Field initializers

As described in [Default values](#), the default value of a struct consists of the value that results from setting all value type fields to their default value and all reference type fields to `null`. For this reason, a struct does not permit instance field declarations to include variable initializers. This restriction applies only to instance fields. Static fields of a struct are permitted to include variable initializers.

The example

```
struct Point
{
    public int x = 1; // Error, initializer not permitted
    public int y = 1; // Error, initializer not permitted
}
```

is in error because the instance field declarations include variable initializers.

Constructors

Unlike a class, a struct is not permitted to declare a parameterless instance constructor. Instead, every struct implicitly has a parameterless instance constructor which always returns the value that results from setting all value type fields to their default value and all reference type fields to null ([Default constructors](#)). A struct can declare instance constructors having parameters. For example

```
struct Point
{
    int x, y;

    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }
}
```

Given the above declaration, the statements

```
Point p1 = new Point();
Point p2 = new Point(0, 0);
```

both create a `Point` with `x` and `y` initialized to zero.

A struct instance constructor is not permitted to include a constructor initializer of the form `base(...)`.

If the struct instance constructor doesn't specify a constructor initializer, the `this` variable corresponds to an `out` parameter of the struct type, and similar to an `out` parameter, `this` must be definitely assigned ([Definite assignment](#)) at every location where the constructor returns. If the struct instance constructor specifies a constructor initializer, the `this` variable corresponds to a `ref` parameter of the struct type, and similar to a `ref` parameter, `this` is considered definitely assigned on entry to the constructor body. Consider the instance constructor implementation below:

```

struct Point
{
    int x, y;

    public int X {
        set { x = value; }
    }

    public int Y {
        set { y = value; }
    }

    public Point(int x, int y) {
        X = x;          // error, this is not yet definitely assigned
        Y = y;          // error, this is not yet definitely assigned
    }
}

```

No instance member function (including the set accessors for the properties `x` and `y`) can be called until all fields of the struct being constructed have been definitely assigned. The only exception involves automatically implemented properties ([Automatically implemented properties](#)). The definite assignment rules ([Simple assignment expressions](#)) specifically exempt assignment to an auto-property of a struct type within an instance constructor of that struct type: such an assignment is considered a definite assignment of the hidden backing field of the auto-property. Thus, the following is allowed:

```

struct Point
{
    public int X { get; set; }
    public int Y { get; set; }

    public Point(int x, int y) {
        X = x; // allowed, definitely assigns backing field
        Y = y; // allowed, definitely assigns backing field
    }
}

```

Destructors

A struct is not permitted to declare a destructor.

Static constructors

Static constructors for structs follow most of the same rules as for classes. The execution of a static constructor for a struct type is triggered by the first of the following events to occur within an application domain:

- A static member of the struct type is referenced.
- An explicitly declared constructor of the struct type is called.

The creation of default values ([Default values](#)) of struct types does not trigger the static constructor. (An example of this is the initial value of elements in an array.)

Struct examples

The following shows two significant examples of using `struct` types to create types that can be used similarly to the predefined types of the language, but with modified semantics.

Database integer type

The `DBInt` struct below implements an integer type that can represent the complete set of values of the `int` type, plus an additional state that indicates an unknown value. A type with these characteristics is commonly used in databases.

```

using System;

public struct DBInt
{
    // The Null member represents an unknown DBInt value.

    public static readonly DBInt Null = new DBInt();

    // When the defined field is true, this DBInt represents a known value
    // which is stored in the value field. When the defined field is false,
    // this DBInt represents an unknown value, and the value field is 0.

    int value;
    bool defined;

    // Private instance constructor. Creates a DBInt with a known value.

    DBInt(int value) {
        this.value = value;
        this.defined = true;
    }

    // The IsNull property is true if this DBInt represents an unknown value.

    public bool IsNull { get { return !defined; } }

    // The Value property is the known value of this DBInt, or 0 if this
    // DBInt represents an unknown value.

    public int Value { get { return value; } }

    // Implicit conversion from int to DBInt.

    public static implicit operator DBInt(int x) {
        return new DBInt(x);
    }

    // Explicit conversion from DBInt to int. Throws an exception if the
    // given DBInt represents an unknown value.

    public static explicit operator int(DBInt x) {
        if (!x.defined) throw new InvalidOperationException();
        return x.value;
    }

    public static DBInt operator +(DBInt x) {
        return x;
    }

    public static DBInt operator -(DBInt x) {
        return x.defined ? -x.value : Null;
    }

    public static DBInt operator +(DBInt x, DBInt y) {
        return x.defined && y.defined? x.value + y.value: Null;
    }

    public static DBInt operator -(DBInt x, DBInt y) {
        return x.defined && y.defined? x.value - y.value: Null;
    }

    public static DBInt operator *(DBInt x, DBInt y) {
        return x.defined && y.defined? x.value * y.value: Null;
    }

    public static DBInt operator /(DBInt x, DBInt y) {
        return x.defined && y.defined? x.value / y.value: Null;
    }
}

```

```

public static DBInt operator %(DBInt x, DBInt y) {
    return x.defined && y.defined? x.value % y.value: Null;
}

public static DBBool operator ==(DBInt x, DBInt y) {
    return x.defined && y.defined? x.value == y.value: DBBool.Null;
}

public static DBBool operator !=(DBInt x, DBInt y) {
    return x.defined && y.defined? x.value != y.value: DBBool.Null;
}

public static DBBool operator >(DBInt x, DBInt y) {
    return x.defined && y.defined? x.value > y.value: DBBool.Null;
}

public static DBBool operator <(DBInt x, DBInt y) {
    return x.defined && y.defined? x.value < y.value: DBBool.Null;
}

public static DBBool operator >=(DBInt x, DBInt y) {
    return x.defined && y.defined? x.value >= y.value: DBBool.Null;
}

public static DBBool operator <=(DBInt x, DBInt y) {
    return x.defined && y.defined? x.value <= y.value: DBBool.Null;
}

public override bool Equals(object obj) {
    if (!(obj is DBInt)) return false;
    DBInt x = (DBInt)obj;
    return value == x.value && defined == x.defined;
}

public override int GetHashCode() {
    return value;
}

public override string ToString() {
    return defined? value.ToString(): "DBInt.Null";
}
}

```

Database boolean type

The `DBBool` struct below implements a three-valued logical type. The possible values of this type are `DBBool.True`, `DBBool.False`, and `DBBool.Null`, where the `Null` member indicates an unknown value. Such three-valued logical types are commonly used in databases.

```

using System;

public struct DBBool
{
    // The three possible DBBool values.

    public static readonly DBBool Null = new DBBool(0);
    public static readonly DBBool False = new DBBool(-1);
    public static readonly DBBool True = new DBBool(1);

    // Private field that stores -1, 0, 1 for False, Null, True.

    sbyte value;

    // Private instance constructor. The value parameter must be -1, 0, or 1.

    DBBool(int value) {

```

```

        this.value = (sbyte)value;
    }

    // Properties to examine the value of a DBBool. Return true if this
    // DBBool has the given value, false otherwise.

    public bool IsNull { get { return value == 0; } }

    public bool IsFalse { get { return value < 0; } }

    public bool IsTrue { get { return value > 0; } }

    // Implicit conversion from bool to DBBool. Maps true to DBBool.True and
    // false to DBBool.False.

    public static implicit operator DBBool(bool x) {
        return x? True: False;
    }

    // Explicit conversion from DBBool to bool. Throws an exception if the
    // given DBBool is Null, otherwise returns true or false.

    public static explicit operator bool(DBBool x) {
        if (x.value == 0) throw new InvalidOperationException();
        return x.value > 0;
    }

    // Equality operator. Returns Null if either operand is Null, otherwise
    // returns True or False.

    public static DBBool operator ==(DBBool x, DBBool y) {
        if (x.value == 0 || y.value == 0) return Null;
        return x.value == y.value? True: False;
    }

    // Inequality operator. Returns Null if either operand is Null, otherwise
    // returns True or False.

    public static DBBool operator !=(DBBool x, DBBool y) {
        if (x.value == 0 || y.value == 0) return Null;
        return x.value != y.value? True: False;
    }

    // Logical negation operator. Returns True if the operand is False, Null
    // if the operand is Null, or False if the operand is True.

    public static DBBool operator !(DBBool x) {
        return new DBBool(-x.value);
    }

    // Logical AND operator. Returns False if either operand is False,
    // otherwise Null if either operand is Null, otherwise True.

    public static DBBool operator &(DBBool x, DBBool y) {
        return new DBBool(x.value < y.value? x.value: y.value);
    }

    // Logical OR operator. Returns True if either operand is True, otherwise
    // Null if either operand is Null, otherwise False.

    public static DBBool operator |(DBBool x, DBBool y) {
        return new DBBool(x.value > y.value? x.value: y.value);
    }

    // Definitely true operator. Returns true if the operand is True, false
    // otherwise.

    public static bool operator true(DBBool x) {
        return x.value > 0;
    }

```

```
        return x.value < 0;
    }

    // Definitely false operator. Returns true if the operand is False, false
    // otherwise.

    public static bool operator false(DBBool x) {
        return x.value < 0;
    }

    public override bool Equals(object obj) {
        if (!(obj is DBBool)) return false;
        return value == ((DBBool)obj).value;
    }

    public override int GetHashCode() {
        return value;
    }

    public override string ToString() {
        if (value > 0) return "DBBool.True";
        if (value < 0) return "DBBool.False";
        return "DBBool.Null";
    }
}
```

Arrays

12/28/2021 • 9 minutes to read • [Edit Online](#)

An array is a data structure that contains a number of variables which are accessed through computed indices. The variables contained in an array, also called the elements of the array, are all of the same type, and this type is called the element type of the array.

An array has a rank which determines the number of indices associated with each array element. The rank of an array is also referred to as the dimensions of the array. An array with a rank of one is called a *single-dimensional array*. An array with a rank greater than one is called a *multi-dimensional array*. Specific sized multi-dimensional arrays are often referred to as two-dimensional arrays, three-dimensional arrays, and so on.

Each dimension of an array has an associated length which is an integral number greater than or equal to zero. The dimension lengths are not part of the type of the array, but rather are established when an instance of the array type is created at run-time. The length of a dimension determines the valid range of indices for that dimension: For a dimension of length `N`, indices can range from `0` to `N - 1` inclusive. The total number of elements in an array is the product of the lengths of each dimension in the array. If one or more of the dimensions of an array have a length of zero, the array is said to be empty.

The element type of an array can be any type, including an array type.

Array types

An array type is written as a *non_array_type* followed by one or more *rank_specifiers*:

```
array_type
: non_array_type rank_specifier+
;

non_array_type
: type
;

rank_specifier
: '[' dim_separator* '['
;

dim_separator
: ','
;
```

A *non_array_type* is any *type* that is not itself an *array_type*.

The rank of an array type is given by the leftmost *rank_specifier* in the *array_type*. A *rank_specifier* indicates that the array is an array with a rank of one plus the number of "`[`", "`,`" tokens in the *rank_specifier*.

The element type of an array type is the type that results from deleting the leftmost *rank_specifier*:

- An array type of the form `T[R]` is an array with rank `R` and a non-array element type `T`.
- An array type of the form `T[R][R1]...[Rn]` is an array with rank `R` and an element type `T[R1]...[Rn]`.

In effect, the *rank_specifiers* are read from left to right before the final non-array element type. The type `int[][,][,]` is a single-dimensional array of three-dimensional arrays of two-dimensional arrays of `int`.

At run-time, a value of an array type can be `null` or a reference to an instance of that array type.

The System.Array type

The type `System.Array` is the abstract base type of all array types. An implicit reference conversion ([Implicit reference conversions](#)) exists from any array type to `System.Array`, and an explicit reference conversion ([Explicit reference conversions](#)) exists from `System.Array` to any array type. Note that `System.Array` is not itself an *array_type*. Rather, it is a *class_type* from which all *array_types* are derived.

At run-time, a value of type `System.Array` can be `null` or a reference to an instance of any array type.

Arrays and the generic IList interface

A one-dimensional array `T[]` implements the interface `System.Collections.Generic.IList<T>` (`IList<T>` for short) and its base interfaces. Accordingly, there is an implicit conversion from `T[]` to `IList<T>` and its base interfaces. In addition, if there is an implicit reference conversion from `S` to `T` then `S[]` implements `IList<T>` and there is an implicit reference conversion from `S[]` to `IList<T>` and its base interfaces ([Implicit reference conversions](#)). If there is an explicit reference conversion from `S` to `T` then there is an explicit reference conversion from `S[]` to `IList<T>` and its base interfaces ([Explicit reference conversions](#)). For example:

```
using System.Collections.Generic;

class Test
{
    static void Main() {
        string[] sa = new string[5];
        object[] oa1 = new object[5];
        object[] oa2 = sa;

        IList<string> lst1 = sa;           // Ok
        IList<string> lst2 = oa1;          // Error, cast needed
        IList<object> lst3 = sa;           // Ok
        IList<object> lst4 = oa1;          // Ok

        IList<string> lst5 = (IList<string>)oa1; // Exception
        IList<string> lst6 = (IList<string>)oa2; // Ok
    }
}
```

The assignment `lst2 = oa1` generates a compile-time error since the conversion from `object[]` to `IList<string>` is an explicit conversion, not implicit. The cast `(IList<string>)oa1` will cause an exception to be thrown at run-time since `oa1` references an `object[]` and not a `string[]`. However the cast `(IList<string>)oa2` will not cause an exception to be thrown since `oa2` references a `string[]`.

Whenever there is an implicit or explicit reference conversion from `S[]` to `IList<T>`, there is also an explicit reference conversion from `IList<T>` and its base interfaces to `S[]` ([Explicit reference conversions](#)).

When an array type `S[]` implements `IList<T>`, some of the members of the implemented interface may throw exceptions. The precise behavior of the implementation of the interface is beyond the scope of this specification.

Array creation

Array instances are created by *array_creation_expressions* ([Array creation expressions](#)) or by field or local variable declarations that include an *array_initializer* ([Array initializers](#)).

When an array instance is created, the rank and length of each dimension are established and then remain constant for the entire lifetime of the instance. In other words, it is not possible to change the rank of an existing array instance, nor is it possible to resize its dimensions.

An array instance is always of an array type. The `System.Array` type is an abstract type that cannot be

instantiated.

Elements of arrays created by *array_creation_expressions* are always initialized to their default value ([Default values](#)).

Array element access

Array elements are accessed using *element_access* expressions ([Array access](#)) of the form `A[I1, I2, ..., In]`, where `A` is an expression of an array type and each `Ix` is an expression of type `int`, `uint`, `long`, `ulong`, or can be implicitly converted to one or more of these types. The result of an array element access is a variable, namely the array element selected by the indices.

The elements of an array can be enumerated using a `foreach` statement ([The foreach statement](#)).

Array members

Every array type inherits the members declared by the `System.Array` type.

Array covariance

For any two *reference_types* `A` and `B`, if an implicit reference conversion ([Implicit reference conversions](#)) or explicit reference conversion ([Explicit reference conversions](#)) exists from `A` to `B`, then the same reference conversion also exists from the array type `A[R]` to the array type `B[R]`, where `R` is any given *rank_specifier* (but the same for both array types). This relationship is known as **array covariance**. Array covariance in particular means that a value of an array type `A[R]` may actually be a reference to an instance of an array type `B[R]`, provided an implicit reference conversion exists from `B` to `A`.

Because of array covariance, assignments to elements of reference type arrays include a run-time check which ensures that the value being assigned to the array element is actually of a permitted type ([Simple assignment](#)). For example:

```
class Test
{
    static void Fill(object[] array, int index, int count, object value) {
        for (int i = index; i < index + count; i++) array[i] = value;
    }

    static void Main() {
        string[] strings = new string[100];
        Fill(strings, 0, 100, "Undefined");
        Fill(strings, 0, 10, null);
        Fill(strings, 90, 10, 0);
    }
}
```

The assignment to `array[i]` in the `Fill` method implicitly includes a run-time check which ensures that the object referenced by `value` is either `null` or an instance that is compatible with the actual element type of `array`. In `Main`, the first two invocations of `Fill` succeed, but the third invocation causes a `System.ArrayTypeMismatchException` to be thrown upon executing the first assignment to `array[i]`. The exception occurs because a boxed `int` cannot be stored in a `string` array.

Array covariance specifically does not extend to arrays of *value_types*. For example, no conversion exists that permits an `int[]` to be treated as an `object[]`.

Array initializers

Array initializers may be specified in field declarations ([Fields](#)), local variable declarations ([Local variable declarations](#)), and array creation expressions ([Array creation expressions](#)):

```
array_initializer
    : '{' variable_initializer_list? '}'
    | '{' variable_initializer_list ',' '}'
    ;

variable_initializer_list
    : variable_initializer (',' variable_initializer)*
    ;

variable_initializer
    : expression
    | array_initializer
    ;
```

An array initializer consists of a sequence of variable initializers, enclosed by "{" and "}" tokens and separated by "," tokens. Each variable initializer is an expression or, in the case of a multi-dimensional array, a nested array initializer.

The context in which an array initializer is used determines the type of the array being initialized. In an array creation expression, the array type immediately precedes the initializer, or is inferred from the expressions in the array initializer. In a field or variable declaration, the array type is the type of the field or variable being declared. When an array initializer is used in a field or variable declaration, such as:

```
int[] a = {0, 2, 4, 6, 8};
```

it is simply shorthand for an equivalent array creation expression:

```
int[] a = new int[] {0, 2, 4, 6, 8};
```

For a single-dimensional array, the array initializer must consist of a sequence of expressions that are assignment compatible with the element type of the array. The expressions initialize array elements in increasing order, starting with the element at index zero. The number of expressions in the array initializer determines the length of the array instance being created. For example, the array initializer above creates an `int[]` instance of length 5 and then initializes the instance with the following values:

```
a[0] = 0; a[1] = 2; a[2] = 4; a[3] = 6; a[4] = 8;
```

For a multi-dimensional array, the array initializer must have as many levels of nesting as there are dimensions in the array. The outermost nesting level corresponds to the leftmost dimension and the innermost nesting level corresponds to the rightmost dimension. The length of each dimension of the array is determined by the number of elements at the corresponding nesting level in the array initializer. For each nested array initializer, the number of elements must be the same as the other array initializers at the same level. The example:

```
int[,] b = {{0, 1}, {2, 3}, {4, 5}, {6, 7}, {8, 9}};
```

creates a two-dimensional array with a length of five for the leftmost dimension and a length of two for the rightmost dimension:

```
int[,] b = new int[5, 2];
```

and then initializes the array instance with the following values:

```
b[0, 0] = 0; b[0, 1] = 1;  
b[1, 0] = 2; b[1, 1] = 3;  
b[2, 0] = 4; b[2, 1] = 5;  
b[3, 0] = 6; b[3, 1] = 7;  
b[4, 0] = 8; b[4, 1] = 9;
```

If a dimension other than the rightmost is given with length zero, the subsequent dimensions are assumed to also have length zero. The example:

```
int[, ] c = {};
```

creates a two-dimensional array with a length of zero for both the leftmost and the rightmost dimension:

```
int[, ] c = new int[0, 0];
```

When an array creation expression includes both explicit dimension lengths and an array initializer, the lengths must be constant expressions and the number of elements at each nesting level must match the corresponding dimension length. Here are some examples:

```
int i = 3;  
int[] x = new int[3] {0, 1, 2};           // OK  
int[] y = new int[i] {0, 1, 2};           // Error, i not a constant  
int[] z = new int[3] {0, 1, 2, 3};         // Error, length/initializer mismatch
```

Here, the initializer for `y` results in a compile-time error because the dimension length expression is not a constant, and the initializer for `z` results in a compile-time error because the length and the number of elements in the initializer do not agree.

Interfaces

12/28/2021 • 29 minutes to read • [Edit Online](#)

An interface defines a contract. A class or struct that implements an interface must adhere to its contract. An interface may inherit from multiple base interfaces, and a class or struct may implement multiple interfaces.

Interfaces can contain methods, properties, events, and indexers. The interface itself does not provide implementations for the members that it defines. The interface merely specifies the members that must be supplied by classes or structs that implement the interface.

Interface declarations

An *interface_declaration* is a *type_declaration* ([Type declarations](#)) that declares a new interface type.

```
interface_declaration
: attributes? interface_modifier* 'partial'? 'interface'
  identifier variant_type_parameter_list? interface_base?
  type_parameter_constraints_clause* interface_body ';'?
```

An *interface_declaration* consists of an optional set of *attributes* ([Attributes](#)), followed by an optional set of *interface_modifiers* ([Interface modifiers](#)), followed by an optional `partial` modifier, followed by the keyword `interface` and an *identifier* that names the interface, followed by an optional *variant_type_parameter_list* specification ([Variant type parameter lists](#)), followed by an optional *interface_base* specification ([Base interfaces](#)), followed by an optional *type_parameter_constraints_clauses* specification ([Type parameter constraints](#)), followed by an *interface_body* ([Interface body](#)), optionally followed by a semicolon.

Interface modifiers

An *interface_declaration* may optionally include a sequence of interface modifiers:

```
interface_modifier
: 'new'
| 'public'
| 'protected'
| 'internal'
| 'private'
| interface_modifier_unsafe
```

It is a compile-time error for the same modifier to appear multiple times in an interface declaration.

The `new` modifier is only permitted on interfaces defined within a class. It specifies that the interface hides an inherited member by the same name, as described in [The new modifier](#).

The `public`, `protected`, `internal`, and `private` modifiers control the accessibility of the interface. Depending on the context in which the interface declaration occurs, only some of these modifiers may be permitted ([Declared accessibility](#)).

Partial modifier

The `partial` modifier indicates that this *interface_declaration* is a partial type declaration. Multiple partial interface declarations with the same name within an enclosing namespace or type declaration combine to form one interface declaration, following the rules specified in [Partial types](#).

Variant type parameter lists

Variant type parameter lists can only occur on interface and delegate types. The difference from ordinary *type_parameter_lists* is the optional *variance_annotation* on each type parameter.

```
variant_type_parameter_list
: '<' variant_type_parameters '>'
;

variant_type_parameters
: attributes? variance_annotation? type_parameter
| variant_type_parameters ',' attributes? variance_annotation? type_parameter
;

variance_annotation
: 'in'
| 'out'
;
```

If the variance annotation is `out`, the type parameter is said to be *covariant*. If the variance annotation is `in`, the type parameter is said to be *contravariant*. If there is no variance annotation, the type parameter is said to be *invariant*.

In the example

```
interface C<out X, in Y, Z>
{
    X M(Y y);
    Z P { get; set; }
}
```

`X` is covariant, `Y` is contravariant and `Z` is invariant.

Variance safety

The occurrence of variance annotations in the type parameter list of a type restricts the places where types can occur within the type declaration.

A type `T` is *output-unsafe* if one of the following holds:

- `T` is a contravariant type parameter
- `T` is an array type with an output-unsafe element type
- `T` is an interface or delegate type `S<A1, ..., Ak>` constructed from a generic type `S<X1, ..., Xk>` where for at least one `Ai` one of the following holds:
 - `Xi` is covariant or invariant and `Ai` is output-unsafe.
 - `Xi` is contravariant or invariant and `Ai` is input-safe.

A type `T` is *input-unsafe* if one of the following holds:

- `T` is a covariant type parameter
- `T` is an array type with an input-unsafe element type
- `T` is an interface or delegate type `S<A1, ..., Ak>` constructed from a generic type `S<X1, ..., Xk>` where for at least one `Ai` one of the following holds:
 - `Xi` is covariant or invariant and `Ai` is input-unsafe.
 - `Xi` is contravariant or invariant and `Ai` is output-unsafe.

Intuitively, an output-unsafe type is prohibited in an output position, and an input-unsafe type is prohibited in an input position.

A type is *output-safe* if it is not output-unsafe, and *input-safe* if it is not input-unsafe.

Variance conversion

The purpose of variance annotations is to provide for more lenient (but still type safe) conversions to interface and delegate types. To this end the definitions of implicit ([Implicit conversions](#)) and explicit conversions ([Explicit conversions](#)) make use of the notion of variance-convertibility, which is defined as follows:

A type $T\langle A_1, \dots, A_n \rangle$ is variance-convertible to a type $T\langle B_1, \dots, B_n \rangle$ if T is either an interface or a delegate type declared with the variant type parameters $T\langle X_1, \dots, X_n \rangle$, and for each variant type parameter X_i one of the following holds:

- X_i is covariant and an implicit reference or identity conversion exists from A_i to B_i
- X_i is contravariant and an implicit reference or identity conversion exists from B_i to A_i
- X_i is invariant and an identity conversion exists from A_i to B_i

Base interfaces

An interface can inherit from zero or more interface types, which are called the *explicit base interfaces* of the interface. When an interface has one or more explicit base interfaces, then in the declaration of that interface, the interface identifier is followed by a colon and a comma separated list of base interface types.

```
interface_base
: ':' interface_type_list
;
```

For a constructed interface type, the explicit base interfaces are formed by taking the explicit base interface declarations on the generic type declaration, and substituting, for each *type_parameter* in the base interface declaration, the corresponding *type_argument* of the constructed type.

The explicit base interfaces of an interface must be at least as accessible as the interface itself ([Accessibility constraints](#)). For example, it is a compile-time error to specify a `private` or `internal` interface in the *interface_base* of a `public` interface.

It is a compile-time error for an interface to directly or indirectly inherit from itself.

The *base interfaces* of an interface are the explicit base interfaces and their base interfaces. In other words, the set of base interfaces is the complete transitive closure of the explicit base interfaces, their explicit base interfaces, and so on. An interface inherits all members of its base interfaces. In the example

```
interface IControl
{
    void Paint();
}

interface ITextBox: IControl
{
    void SetText(string text);
}

interface IListBox: IControl
{
    void SetItems(string[] items);
}

interface IComboBox: ITextBox, IListBox {}
```

the base interfaces of `IComboBox` are `IControl`, `ITextBox`, and `IListBox`.

In other words, the `IComboBox` interface above inherits members `SetText` and `SetItems` as well as `Paint`.

Every base interface of an interface must be output-safe ([Variance safety](#)). A class or struct that implements an interface also implicitly implements all of the interface's base interfaces.

Interface body

The *interface_body* of an interface defines the members of the interface.

```
interface_body
: '{' interface_member_declaration* '}'
;
```

Interface members

The members of an interface are the members inherited from the base interfaces and the members declared by the interface itself.

```
interface_member_declaration
: interface_method_declaration
| interface_property_declaration
| interface_event_declaration
| interface_indexer_declaration
;
```

An interface declaration may declare zero or more members. The members of an interface must be methods, properties, events, or indexers. An interface cannot contain constants, fields, operators, instance constructors, destructors, or types, nor can an interface contain static members of any kind.

All interface members implicitly have public access. It is a compile-time error for interface member declarations to include any modifiers. In particular, interfaces members cannot be declared with the modifiers `abstract`, `public`, `protected`, `internal`, `private`, `virtual`, `override`, or `static`.

The example

```
public delegate void StringListEvent(IStringList sender);

public interface IStringList
{
    void Add(string s);
    int Count { get; }
    event StringListEvent Changed;
    string this[int index] { get; set; }
}
```

declares an interface that contains one each of the possible kinds of members: A method, a property, an event, and an indexer.

An *interface_declaration* creates a new declaration space ([Declarations](#)), and the *interface_member_declarations* immediately contained by the *interface_declaration* introduce new members into this declaration space. The following rules apply to *interface_member_declarations*:

- The name of a method must differ from the names of all properties and events declared in the same interface. In addition, the signature ([Signatures and overloading](#)) of a method must differ from the signatures of all other methods declared in the same interface, and two methods declared in the same interface may not have signatures that differ solely by `ref` and `out`.
- The name of a property or event must differ from the names of all other members declared in the same interface.
- The signature of an indexer must differ from the signatures of all other indexers declared in the same

interface.

The inherited members of an interface are specifically not part of the declaration space of the interface. Thus, an interface is allowed to declare a member with the same name or signature as an inherited member. When this occurs, the derived interface member is said to hide the base interface member. Hiding an inherited member is not considered an error, but it does cause the compiler to issue a warning. To suppress the warning, the declaration of the derived interface member must include a `new` modifier to indicate that the derived member is intended to hide the base member. This topic is discussed further in [Hiding through inheritance](#).

If a `new` modifier is included in a declaration that doesn't hide an inherited member, a warning is issued to that effect. This warning is suppressed by removing the `new` modifier.

Note that the members in class `object` are not, strictly speaking, members of any interface ([Interface members](#)). However, the members in class `object` are available via member lookup in any interface type ([Member lookup](#)).

Interface methods

Interface methods are declared using *interface_method_declarations*:

```
interface_method_declaration
: attributes? 'new'? return_type identifier type_parameter_list
  '(' formal_parameter_list? ')' type_parameter_constraints_clause* ';'
;
```

The *attributes*, *return_type*, *identifier*, and *formal_parameter_list* of an interface method declaration have the same meaning as those of a method declaration in a class ([Methods](#)). An interface method declaration is not permitted to specify a method body, and the declaration therefore always ends with a semicolon.

Each formal parameter type of an interface method must be input-safe ([Variance safety](#)), and the return type must be either `void` or output-safe. Furthermore, each class type constraint, interface type constraint and type parameter constraint on any type parameter of the method must be input-safe.

These rules ensure that any covariant or contravariant usage of the interface remains type-safe. For example,

```
interface I<out T> { void M<U>() where U : T; }
```

is illegal because the usage of `T` as a type parameter constraint on `U` is not input-safe.

Were this restriction not in place it would be possible to violate type safety in the following manner:

```
class B {}
class D : B {}
class E : B {}
class C : I<D> { public void M<U>() {...} }
...
I<B> b = new C();
b.M<E>();
```

This is actually a call to `C.M<E>`. But that call requires that `E` derive from `D`, so type safety would be violated here.

Interface properties

Interface properties are declared using *interface_property_declarations*:


```

interface_property_declaration
: attributes? 'new'? type identifier '{' interface_accessors '}'
;

interface_accessors
: attributes? 'get' ';'
| attributes? 'set' ';'
| attributes? 'get' ';' attributes? 'set' ';'
| attributes? 'set' ';' attributes? 'get' ';'
;

```

The *attributes*, *type*, and *identifier* of an interface property declaration have the same meaning as those of a property declaration in a class ([Properties](#)).

The accessors of an interface property declaration correspond to the accessors of a class property declaration ([Accessors](#)), except that the accessor body must always be a semicolon. Thus, the accessors simply indicate whether the property is read-write, read-only, or write-only.

The type of an interface property must be output-safe if there is a get accessor, and must be input-safe if there is a set accessor.

Interface events

Interface events are declared using *interface_event_declarations*:

```

interface_event_declaration
: attributes? 'new'? 'event' type identifier ';'
;

```

The *attributes*, *type*, and *identifier* of an interface event declaration have the same meaning as those of an event declaration in a class ([Events](#)).

The type of an interface event must be input-safe.

Interface indexers

Interface indexers are declared using *interface_indexer_declarations*:

```

interface_indexer_declaration
: attributes? 'new'? type 'this' '[' formal_parameter_list ']' '{' interface_accessors '}'
;

```

The *attributes*, *type*, and *formal_parameter_list* of an interface indexer declaration have the same meaning as those of an indexer declaration in a class ([Indexers](#)).

The accessors of an interface indexer declaration correspond to the accessors of a class indexer declaration ([Indexers](#)), except that the accessor body must always be a semicolon. Thus, the accessors simply indicate whether the indexer is read-write, read-only, or write-only.

All the formal parameter types of an interface indexer must be input-safe. In addition, any `out` or `ref` formal parameter types must also be output-safe. Note that even `out` parameters are required to be input-safe, due to a limitation of the underlying execution platform.

The type of an interface indexer must be output-safe if there is a get accessor, and must be input-safe if there is a set accessor.

Interface member access

Interface members are accessed through member access ([Member access](#)) and indexer access ([Indexer access](#)) expressions of the form `I.M` and `I[A]`, where `I` is an interface type, `M` is a method, property, or event of that

interface type, and `A` is an indexer argument list.

For interfaces that are strictly single-inheritance (each interface in the inheritance chain has exactly zero or one direct base interface), the effects of the member lookup ([Member lookup](#)), method invocation ([Method invocations](#)), and indexer access ([Indexer access](#)) rules are exactly the same as for classes and structs: More derived members hide less derived members with the same name or signature. However, for multiple-inheritance interfaces, ambiguities can occur when two or more unrelated base interfaces declare members with the same name or signature. This section shows several examples of such situations. In all cases, explicit casts can be used to resolve the ambiguities.

In the example

```
interface IList
{
    int Count { get; set; }
}

interface ICounter
{
    void Count(int i);
}

interface IListCounter: IList, ICounter {}

class C
{
    void Test(IListCounter x) {
        x.Count(1);           // Error
        x.Count = 1;          // Error
        ((IList)x).Count = 1; // Ok, invokes IList.Count.set
        ((ICounter)x).Count(1); // Ok, invokes ICounter.Count
    }
}
```

the first two statements cause compile-time errors because the member lookup ([Member lookup](#)) of `Count` in `IListCounter` is ambiguous. As illustrated by the example, the ambiguity is resolved by casting `x` to the appropriate base interface type. Such casts have no run-time costs—they merely consist of viewing the instance as a less derived type at compile-time.

In the example

```

interface IInteger
{
    void Add(int i);
}

interface IDouble
{
    void Add(double d);
}

interface INumber: IInteger, IDouble {}

class C
{
    void Test(INumber n) {
        n.Add(1);           // Invokes IInteger.Add
        n.Add(1.0);         // Only IDouble.Add is applicable
        ((IInteger)n).Add(1); // Only IInteger.Add is a candidate
        ((IDouble)n).Add(1);  // Only IDouble.Add is a candidate
    }
}

```

the invocation `n.Add(1)` selects `IInteger.Add` by applying the overload resolution rules of [Overload resolution](#). Similarly the invocation `n.Add(1.0)` selects `IDouble.Add`. When explicit casts are inserted, there is only one candidate method, and thus no ambiguity.

In the example

```

interface IBase
{
    void F(int i);
}

interface ILeft: IBase
{
    new void F(int i);
}

interface IRight: IBase
{
    void G();
}

interface IDerived: ILeft, IRight {}

class A
{
    void Test(IDerived d) {
        d.F(1);           // Invokes ILeft.F
        ((IBase)d).F(1);   // Invokes IBase.F
        ((ILeft)d).F(1);   // Invokes ILeft.F
        ((IRight)d).F(1);  // Invokes IBase.F
    }
}

```

the `IBase.F` member is hidden by the `ILeft.F` member. The invocation `d.F(1)` thus selects `ILeft.F`, even though `IBase.F` appears to not be hidden in the access path that leads through `IRight`.

The intuitive rule for hiding in multiple-inheritance interfaces is simply this: If a member is hidden in any access path, it is hidden in all access paths. Because the access path from `IDerived` to `ILeft` to `IBase` hides `IBase.F`, the member is also hidden in the access path from `IDerived` to `IRight` to `IBase`.

Fully qualified interface member names

An interface member is sometimes referred to by its *fully qualified name*. The fully qualified name of an interface member consists of the name of the interface in which the member is declared, followed by a dot, followed by the name of the member. The fully qualified name of a member references the interface in which the member is declared. For example, given the declarations

```
interface IControl
{
    void Paint();
}

interface ITextBox: IControl
{
    void SetText(string text);
}
```

the fully qualified name of `Paint` is `IControl.Paint` and the fully qualified name of `SetText` is `ITextBox.SetText`.

In the example above, it is not possible to refer to `Paint` as `ITextBox.Paint`.

When an interface is part of a namespace, the fully qualified name of an interface member includes the namespace name. For example

```
namespace System
{
    public interface ICloneable
    {
        object Clone();
    }
}
```

Here, the fully qualified name of the `Clone` method is `System.ICloneable.Clone`.

Interface implementations

Interfaces may be implemented by classes and structs. To indicate that a class or struct directly implements an interface, the interface identifier is included in the base class list of the class or struct. For example:

```
interface ICloneable
{
    object Clone();
}

interface IComparable
{
    int CompareTo(object other);
}

class ListEntry: ICloneable, IComparable
{
    public object Clone() {...}
    public int CompareTo(object other) {...}
}
```

A class or struct that directly implements an interface also directly implements all of the interface's base interfaces implicitly. This is true even if the class or struct doesn't explicitly list all base interfaces in the base class list. For example:

```

interface IControl
{
    void Paint();
}

interface ITextBox: IControl
{
    void SetText(string text);
}

class TextBox: ITextBox
{
    public void Paint() {...}
    public void SetText(string text) {...}
}

```

Here, class `TextBox` implements both `IControl` and `ITextBox`.

When a class `C` directly implements an interface, all classes derived from `C` also implement the interface implicitly. The base interfaces specified in a class declaration can be constructed interface types ([Constructed types](#)). A base interface cannot be a type parameter on its own, though it can involve the type parameters that are in scope. The following code illustrates how a class can implement and extend constructed types:

```

class C<U,V> {}

interface I1<V> {}

class D: C<string,int>, I1<string> {}

class E<T>: C<int,T>, I1<T> {}

```

The base interfaces of a generic class declaration must satisfy the uniqueness rule described in [Uniqueness of implemented interfaces](#).

Explicit interface member implementations

For purposes of implementing interfaces, a class or struct may declare *explicit interface member implementations*. An explicit interface member implementation is a method, property, event, or indexer declaration that references a fully qualified interface member name. For example

```

interface IList<T>
{
    T[] GetElements();
}

interface IDictionary<K,V>
{
    V this[K key];
    void Add(K key, V value);
}

class List<T>: IList<T>, IDictionary<int,T>
{
    T[] IList<T>.GetElements() {...}
    T IDictionary<int,T>.this[int index] {...}
    void IDictionary<int,T>.Add(int index, T value) {...}
}

```

Here `IDictionary<int,T>.this` and `IDictionary<int,T>.Add` are explicit interface member implementations.

In some cases, the name of an interface member may not be appropriate for the implementing class, in which case the interface member may be implemented using explicit interface member implementation. A class implementing a file abstraction, for example, would likely implement a `Close` member function that has the effect of releasing the file resource, and implement the `Dispose` method of the `IDisposable` interface using explicit interface member implementation:

```
interface IDisposable
{
    void Dispose();
}

class MyFile: IDisposable
{
    void IDisposable.Dispose() {
        Close();
    }

    public void Close() {
        // Do what's necessary to close the file
        System.GC.SuppressFinalize(this);
    }
}
```

It is not possible to access an explicit interface member implementation through its fully qualified name in a method invocation, property access, or indexer access. An explicit interface member implementation can only be accessed through an interface instance, and is in that case referenced simply by its member name.

It is a compile-time error for an explicit interface member implementation to include access modifiers, and it is a compile-time error to include the modifiers `abstract`, `virtual`, `override`, or `static`.

Explicit interface member implementations have different accessibility characteristics than other members. Because explicit interface member implementations are never accessible through their fully qualified name in a method invocation or a property access, they are in a sense private. However, since they can be accessed through an interface instance, they are in a sense also public.

Explicit interface member implementations serve two primary purposes:

- Because explicit interface member implementations are not accessible through class or struct instances, they allow interface implementations to be excluded from the public interface of a class or struct. This is particularly useful when a class or struct implements an internal interface that is of no interest to a consumer of that class or struct.
- Explicit interface member implementations allow disambiguation of interface members with the same signature. Without explicit interface member implementations it would be impossible for a class or struct to have different implementations of interface members with the same signature and return type, as would it be impossible for a class or struct to have any implementation at all of interface members with the same signature but with different return types.

For an explicit interface member implementation to be valid, the class or struct must name an interface in its base class list that contains a member whose fully qualified name, type, and parameter types exactly match those of the explicit interface member implementation. Thus, in the following class

```
class Shape: ICloneable
{
    object ICloneable.Clone() {...}
    int IComparable.CompareTo(object other) {...}    // invalid
}
```

the declaration of `Comparable.CompareTo` results in a compile-time error because `Comparable` is not listed in the base class list of `Shape` and is not a base interface of `ICloneable`. Likewise, in the declarations

```
class Shape: ICloneable
{
    object ICloneable.Clone() {...}
}

class Ellipse: Shape
{
    object ICloneable.Clone() {...}    // invalid
}
```

the declaration of `ICloneable.Clone` in `Ellipse` results in a compile-time error because `ICloneable` is not explicitly listed in the base class list of `Ellipse`.

The fully qualified name of an interface member must reference the interface in which the member was declared. Thus, in the declarations

```
interface IControl
{
    void Paint();
}

interface ITextBox: IControl
{
    void SetText(string text);
}

class TextBox: ITextBox
{
    void IControl.Paint() {...}
    void ITextBox.SetText(string text) {...}
}
```

the explicit interface member implementation of `Paint` must be written as `IControl.Paint`.

Uniqueness of implemented interfaces

The interfaces implemented by a generic type declaration must remain unique for all possible constructed types. Without this rule, it would be impossible to determine the correct method to call for certain constructed types. For example, suppose a generic class declaration were permitted to be written as follows:

```
interface I<T>
{
    void F();
}

class X<U,V>: I<U>, I<V>                // Error: I<U> and I<V> conflict
{
    void I<U>.F() {...}
    void I<V>.F() {...}
}
```

Were this permitted, it would be impossible to determine which code to execute in the following case:

```
I<int> x = new X<int,int>();
x.F();
```

To determine if the interface list of a generic type declaration is valid, the following steps are performed:

- Let `L` be the list of interfaces directly specified in a generic class, struct, or interface declaration `C`.
- Add to `L` any base interfaces of the interfaces already in `L`.
- Remove any duplicates from `L`.
- If any possible constructed type created from `C` would, after type arguments are substituted into `L`, cause two interfaces in `L` to be identical, then the declaration of `C` is invalid. Constraint declarations are not considered when determining all possible constructed types.

In the class declaration `x` above, the interface list `L` consists of `I<U>` and `I<V>`. The declaration is invalid because any constructed type with `u` and `v` being the same type would cause these two interfaces to be identical types.

It is possible for interfaces specified at different inheritance levels to unify:

```
interface I<T>
{
    void F();
}

class Base<U>: I<U>
{
    void I<U>.F() {...}
}

class Derived<U,V>: Base<U>, I<V>    // Ok
{
    void I<V>.F() {...}
}
```

This code is valid even though `Derived<U,V>` implements both `I<U>` and `I<V>`. The code

```
I<int> x = new Derived<int,int>();
x.F();
```

invokes the method in `Derived`, since `Derived<int,int>` effectively re-implements `I<int>` ([Interface re-implementation](#)).

Implementation of generic methods

When a generic method implicitly implements an interface method, the constraints given for each method type parameter must be equivalent in both declarations (after any interface type parameters are replaced with the appropriate type arguments), where method type parameters are identified by ordinal positions, left to right.

When a generic method explicitly implements an interface method, however, no constraints are allowed on the implementing method. Instead, the constraints are inherited from the interface method


```

interface I<A,B,C>
{
    void F<T>(T t) where T: A;
    void G<T>(T t) where T: B;
    void H<T>(T t) where T: C;
}

class C: I<object,C,string>
{
    public void F<T>(T t) {...}           // Ok
    public void G<T>(T t) where T: C {...} // Ok
    public void H<T>(T t) where T: string {...} // Error
}

```

The method `C.F<T>` implicitly implements `I<object,C,string>.F<T>`. In this case, `C.F<T>` is not required (nor permitted) to specify the constraint `T:object` since `object` is an implicit constraint on all type parameters. The method `C.G<T>` implicitly implements `I<object,C,string>.G<T>` because the constraints match those in the interface, after the interface type parameters are replaced with the corresponding type arguments. The constraint for method `C.H<T>` is an error because sealed types (`string` in this case) cannot be used as constraints. Omitting the constraint would also be an error since constraints of implicit interface method implementations are required to match. Thus, it is impossible to implicitly implement `I<object,C,string>.H<T>`. This interface method can only be implemented using an explicit interface member implementation:

```

class C: I<object,C,string>
{
    ...

    public void H<U>(U u) where U: class {...}

    void I<object,C,string>.H<T>(T t) {
        string s = t;    // Ok
        H<T>(t);
    }
}

```

In this example, the explicit interface member implementation invokes a public method having strictly weaker constraints. Note that the assignment from `t` to `s` is valid since `T` inherits a constraint of `T:string`, even though this constraint is not expressible in source code.

Interface mapping

A class or struct must provide implementations of all members of the interfaces that are listed in the base class list of the class or struct. The process of locating implementations of interface members in an implementing class or struct is known as *interface mapping*.

Interface mapping for a class or struct `C` locates an implementation for each member of each interface specified in the base class list of `C`. The implementation of a particular interface member `I.M`, where `I` is the interface in which the member `M` is declared, is determined by examining each class or struct `S`, starting with `C` and repeating for each successive base class of `C`, until a match is located:

- If `S` contains a declaration of an explicit interface member implementation that matches `I` and `M`, then this member is the implementation of `I.M`.
- Otherwise, if `S` contains a declaration of a non-static public member that matches `M`, then this member is the implementation of `I.M`. If more than one member matches, it is unspecified which member is the implementation of `I.M`. This situation can only occur if `S` is a constructed type where the two members as declared in the generic type have different signatures, but the type arguments make their signatures identical.

A compile-time error occurs if implementations cannot be located for all members of all interfaces specified in the base class list of `C`. Note that the members of an interface include those members that are inherited from base interfaces.

For purposes of interface mapping, a class member `A` matches an interface member `B` when:

- `A` and `B` are methods, and the name, type, and formal parameter lists of `A` and `B` are identical.
- `A` and `B` are properties, the name and type of `A` and `B` are identical, and `A` has the same accessors as `B` (`A` is permitted to have additional accessors if it is not an explicit interface member implementation).
- `A` and `B` are events, and the name and type of `A` and `B` are identical.
- `A` and `B` are indexers, the type and formal parameter lists of `A` and `B` are identical, and `A` has the same accessors as `B` (`A` is permitted to have additional accessors if it is not an explicit interface member implementation).

Notable implications of the interface mapping algorithm are:

- Explicit interface member implementations take precedence over other members in the same class or struct when determining the class or struct member that implements an interface member.
- Neither non-public nor static members participate in interface mapping.

In the example

```
interface ICloneable
{
    object Clone();
}

class C: ICloneable
{
    object ICloneable.Clone() {...}
    public object Clone() {...}
}
```

the `ICloneable.Clone` member of `C` becomes the implementation of `Clone` in `ICloneable` because explicit interface member implementations take precedence over other members.

If a class or struct implements two or more interfaces containing a member with the same name, type, and parameter types, it is possible to map each of those interface members onto a single class or struct member. For example

```
interface IControl
{
    void Paint();
}

interface IForm
{
    void Paint();
}

class Page: IControl, IForm
{
    public void Paint() {...}
}
```

Here, the `Paint` methods of both `IControl` and `IForm` are mapped onto the `Paint` method in `Page`. It is of course also possible to have separate explicit interface member implementations for the two methods.

If a class or struct implements an interface that contains hidden members, then some members must necessarily be implemented through explicit interface member implementations. For example

```
interface IBase
{
    int P { get; }
}

interface IDerived: IBase
{
    new int P();
}
```

An implementation of this interface would require at least one explicit interface member implementation, and would take one of the following forms

```
class C: IDerived
{
    int IBase.P { get {...} }
    int IDerived.P() {...}
}

class C: IDerived
{
    public int P { get {...} }
    int IDerived.P() {...}
}

class C: IDerived
{
    int IBase.P { get {...} }
    public int P() {...}
}
```

When a class implements multiple interfaces that have the same base interface, there can be only one implementation of the base interface. In the example

```
interface IControl
{
    void Paint();
}

interface ITextBox: IControl
{
    void SetText(string text);
}

interface IListBox: IControl
{
    void SetItems(string[] items);
}

class ComboBox: IControl, ITextBox, IListBox
{
    void IControl.Paint() {...}
    void ITextBox.SetText(string text) {...}
    void IListBox.SetItems(string[] items) {...}
}
```

it is not possible to have separate implementations for the `IControl` named in the base class list, the `IControl` inherited by `ITextBox`, and the `IControl` inherited by `IListBox`. Indeed, there is no notion of a separate

identity for these interfaces. Rather, the implementations of `ITextBox` and `IListBox` share the same implementation of `IControl`, and `ComboBox` is simply considered to implement three interfaces, `IControl`, `ITextBox`, and `IListBox`.

The members of a base class participate in interface mapping. In the example

```
interface Interface1
{
    void F();
}

class Class1
{
    public void F() {}
    public void G() {}
}

class Class2: Class1, Interface1
{
    new public void G() {}
}
```

the method `F` in `Class1` is used in `Class2`'s implementation of `Interface1`.

Interface implementation inheritance

A class inherits all interface implementations provided by its base classes.

Without explicitly *re-implementing* an interface, a derived class cannot in any way alter the interface mappings it inherits from its base classes. For example, in the declarations

```
interface IControl
{
    void Paint();
}

class Control: IControl
{
    public void Paint() {...}
}

class TextBox: Control
{
    new public void Paint() {...}
}
```

the `Paint` method in `TextBox` hides the `Paint` method in `Control`, but it does not alter the mapping of `Control.Paint` onto `IControl.Paint`, and calls to `Paint` through class instances and interface instances will have the following effects

```
Control c = new Control();
TextBox t = new TextBox();
IControl ic = c;
IControl it = t;

c.Paint();           // invokes Control.Paint();
t.Paint();           // invokes TextBox.Paint();
ic.Paint();          // invokes Control.Paint();
it.Paint();          // invokes Control.Paint();
```

However, when an interface method is mapped onto a virtual method in a class, it is possible for derived classes to override the virtual method and alter the implementation of the interface. For example, rewriting the

declarations above to

```
interface IControl
{
    void Paint();
}

class Control: IControl
{
    public virtual void Paint() {...}
}

class TextBox: Control
{
    public override void Paint() {...}
}
```

the following effects will now be observed

```
Control c = new Control();
TextBox t = new TextBox();
IControl ic = c;
IControl it = t;
c.Paint();           // invokes Control.Paint();
t.Paint();           // invokes TextBox.Paint();
ic.Paint();          // invokes Control.Paint();
it.Paint();          // invokes TextBox.Paint();
```

Since explicit interface member implementations cannot be declared virtual, it is not possible to override an explicit interface member implementation. However, it is perfectly valid for an explicit interface member implementation to call another method, and that other method can be declared virtual to allow derived classes to override it. For example

```
interface IControl
{
    void Paint();
}

class Control: IControl
{
    void IControl.Paint() { PaintControl(); }
    protected virtual void PaintControl() {...}
}

class TextBox: Control
{
    protected override void PaintControl() {...}
}
```

Here, classes derived from `Control` can specialize the implementation of `IControl.Paint` by overriding the `PaintControl` method.

Interface re-implementation

A class that inherits an interface implementation is permitted to *re-implement* the interface by including it in the base class list.

A re-implementation of an interface follows exactly the same interface mapping rules as an initial implementation of an interface. Thus, the inherited interface mapping has no effect whatsoever on the interface mapping established for the re-implementation of the interface. For example, in the declarations

```

interface IControl
{
    void Paint();
}

class Control: IControl
{
    void IControl.Paint() {...}
}

class MyControl: Control, IControl
{
    public void Paint() {}
}

```

the fact that `Control` maps `IControl.Paint` onto `Control.IControl.Paint` doesn't affect the re-implementation in `MyControl`, which maps `IControl.Paint` onto `MyControl.Paint`.

Inherited public member declarations and inherited explicit interface member declarations participate in the interface mapping process for re-implemented interfaces. For example

```

interface IMethods
{
    void F();
    void G();
    void H();
    void I();
}

class Base: IMethods
{
    void IMethods.F() {}
    void IMethods.G() {}
    public void H() {}
    public void I() {}
}

class Derived: Base, IMethods
{
    public void F() {}
    void IMethods.H() {}
}

```

Here, the implementation of `IMethods` in `Derived` maps the interface methods onto `Derived.F`, `Base.IMethods.G`, `Derived.IMethods.H`, and `Base.I`.

When a class implements an interface, it implicitly also implements all of that interface's base interfaces. Likewise, a re-implementation of an interface is also implicitly a re-implementation of all of the interface's base interfaces. For example

```

interface IBase
{
    void F();
}

interface IDerived: IBase
{
    void G();
}

class C: IDerived
{
    void IBase.F() {...}
    void IDerived.G() {...}
}

class D: C, IDerived
{
    public void F() {...}
    public void G() {...}
}

```

Here, the re-implementation of `IDerived` also re-implements `IBase`, mapping `IBase.F` onto `D.F`.

Abstract classes and interfaces

Like a non-abstract class, an abstract class must provide implementations of all members of the interfaces that are listed in the base class list of the class. However, an abstract class is permitted to map interface methods onto abstract methods. For example

```

interface IMethods
{
    void F();
    void G();
}

abstract class C: IMethods
{
    public abstract void F();
    public abstract void G();
}

```

Here, the implementation of `IMethods` maps `F` and `G` onto abstract methods, which must be overridden in non-abstract classes that derive from `C`.

Note that explicit interface member implementations cannot be abstract, but explicit interface member implementations are of course permitted to call abstract methods. For example

```

interface IMethods
{
    void F();
    void G();
}

abstract class C: IMethods
{
    void IMethods.F() { FF(); }
    void IMethods.G() { GG(); }
    protected abstract void FF();
    protected abstract void GG();
}

```

Here, non-abstract classes that derive from `C` would be required to override `FF` and `GG`, thus providing the actual implementation of `IMethods`.

Enums

12/28/2021 • 6 minutes to read • [Edit Online](#)

An **enum type** is a distinct value type ([Value types](#)) that declares a set of named constants.

The example

```
enum Color
{
    Red,
    Green,
    Blue
}
```

declares an enum type named `Color` with members `Red`, `Green`, and `Blue`.

Enum declarations

An enum declaration declares a new enum type. An enum declaration begins with the keyword `enum`, and defines the name, accessibility, underlying type, and members of the enum.

```
enum_declaration
: attributes? enum_modifier* 'enum' identifier enum_base? enum_body ';'
;

enum_base
: ':' integral_type
;

enum_body
: '{' enum_member_declarations? '}'
| '{' enum_member_declarations ',' '}'
;
```

Each enum type has a corresponding integral type called the **underlying type** of the enum type. This underlying type must be able to represent all the enumerator values defined in the enumeration. An enum declaration may explicitly declare an underlying type of `byte`, `sbyte`, `short`, `ushort`, `int`, `uint`, `long` or `ulong`. Note that `char` cannot be used as an underlying type. An enum declaration that does not explicitly declare an underlying type has an underlying type of `int`.

The example

```
enum Color: long
{
    Red,
    Green,
    Blue
}
```

declares an enum with an underlying type of `long`. A developer might choose to use an underlying type of `long`, as in the example, to enable the use of values that are in the range of `long` but not in the range of `int`, or to preserve this option for the future.

Enum modifiers

An *enum_declaration* may optionally include a sequence of enum modifiers:

```
enum_modifier
: 'new'
| 'public'
| 'protected'
| 'internal'
| 'private'
;
```

It is a compile-time error for the same modifier to appear multiple times in an enum declaration.

The modifiers of an enum declaration have the same meaning as those of a class declaration ([Class modifiers](#)).

Note, however, that the `abstract` and `sealed` modifiers are not permitted in an enum declaration. Enums cannot be abstract and do not permit derivation.

Enum members

The body of an enum type declaration defines zero or more enum members, which are the named constants of the enum type. No two enum members can have the same name.

```
enum_member_declarations
: enum_member_declaration (',' enum_member_declaration)*
;

enum_member_declaration
: attributes? identifier ('=' constant_expression)?
;
```

Each enum member has an associated constant value. The type of this value is the underlying type for the containing enum. The constant value for each enum member must be in the range of the underlying type for the enum. The example

```
enum Color: uint
{
    Red = -1,
    Green = -2,
    Blue = -3
}
```

results in a compile-time error because the constant values `-1`, `-2`, and `-3` are not in the range of the underlying integral type `uint`.

Multiple enum members may share the same associated value. The example

```
enum Color
{
    Red,
    Green,
    Blue,

    Max = Blue
}
```

shows an enum in which two enum members -- `Blue` and `Max` -- have the same associated value.

The associated value of an enum member is assigned either implicitly or explicitly. If the declaration of the enum member has a *constant_expression* initializer, the value of that constant expression, implicitly converted to the underlying type of the enum, is the associated value of the enum member. If the declaration of the enum member has no initializer, its associated value is set implicitly, as follows:

- If the enum member is the first enum member declared in the enum type, its associated value is zero.
- Otherwise, the associated value of the enum member is obtained by increasing the associated value of the textually preceding enum member by one. This increased value must be within the range of values that can be represented by the underlying type, otherwise a compile-time error occurs.

The example

```
using System;

enum Color
{
    Red,
    Green = 10,
    Blue
}

class Test
{
    static void Main() {
        Console.WriteLine(StringFromColor(Color.Red));
        Console.WriteLine(StringFromColor(Color.Green));
        Console.WriteLine(StringFromColor(Color.Blue));
    }

    static string StringFromColor(Color c) {
        switch (c) {
            case Color.Red:
                return String.Format("Red = {0}", (int) c);

            case Color.Green:
                return String.Format("Green = {0}", (int) c);

            case Color.Blue:
                return String.Format("Blue = {0}", (int) c);

            default:
                return "Invalid color";
        }
    }
}
```

prints out the enum member names and their associated values. The output is:

```
Red = 0
Green = 10
Blue = 11
```

for the following reasons:

- the enum member `Red` is automatically assigned the value zero (since it has no initializer and is the first enum member);
- the enum member `Green` is explicitly given the value `10`;
- and the enum member `Blue` is automatically assigned the value one greater than the member that textually precedes it.

The associated value of an enum member may not, directly or indirectly, use the value of its own associated enum member. Other than this circularity restriction, enum member initializers may freely refer to other enum member initializers, regardless of their textual position. Within an enum member initializer, values of other enum members are always treated as having the type of their underlying type, so that casts are not necessary when referring to other enum members.

The example

```
enum Circular
{
    A = B,
    B
}
```

results in a compile-time error because the declarations of `A` and `B` are circular. `A` depends on `B` explicitly, and `B` depends on `A` implicitly.

Enum members are named and scoped in a manner exactly analogous to fields within classes. The scope of an enum member is the body of its containing enum type. Within that scope, enum members can be referred to by their simple name. From all other code, the name of an enum member must be qualified with the name of its enum type. Enum members do not have any declared accessibility -- an enum member is accessible if its containing enum type is accessible.

The System.Enum type

The type `System.Enum` is the abstract base class of all enum types (this is distinct and different from the underlying type of the enum type), and the members inherited from `System.Enum` are available in any enum type. A boxing conversion ([Boxing conversions](#)) exists from any enum type to `System.Enum`, and an unboxing conversion ([Unboxing conversions](#)) exists from `System.Enum` to any enum type.

Note that `System.Enum` is not itself an *enum_type*. Rather, it is a *class_type* from which all *enum_types* are derived. The type `System.Enum` inherits from the type `System.ValueType` ([The System.ValueType type](#)), which, in turn, inherits from type `object`. At run-time, a value of type `System.Enum` can be `null` or a reference to a boxed value of any enum type.

Enum values and operations

Each enum type defines a distinct type; an explicit enumeration conversion ([Explicit enumeration conversions](#)) is required to convert between an enum type and an integral type, or between two enum types. The set of values that an enum type can take on is not limited by its enum members. In particular, any value of the underlying type of an enum can be cast to the enum type, and is a distinct valid value of that enum type.

Enum members have the type of their containing enum type (except within other enum member initializers: see [Enum members](#)). The value of an enum member declared in enum type `E` with associated value `v` is `(E)v`.

The following operators can be used on values of enum types: `==`, `!=`, `<`, `>`, `<=`, `>=` ([Enumeration comparison operators](#)), binary `+` ([Addition operator](#)), binary `-` ([Subtraction operator](#)), `^`, `&`, `|` ([Enumeration logical operators](#)), `~` ([Bitwise complement operator](#)), `++` and `--` ([Postfix increment and decrement operators](#) and [Prefix increment and decrement operators](#)).

Every enum type automatically derives from the class `System.Enum` (which, in turn, derives from `System.ValueType` and `object`). Thus, inherited methods and properties of this class can be used on values of an enum type.

Delegates

12/28/2021 • 9 minutes to read • [Edit Online](#)

Delegates enable scenarios that other languages—such as C++, Pascal, and Modula -- have addressed with function pointers. Unlike C++ function pointers, however, delegates are fully object oriented, and unlike C++ pointers to member functions, delegates encapsulate both an object instance and a method.

A delegate declaration defines a class that is derived from the class `System.Delegate`. A delegate instance encapsulates an invocation list, which is a list of one or more methods, each of which is referred to as a callable entity. For instance methods, a callable entity consists of an instance and a method on that instance. For static methods, a callable entity consists of just a method. Invoking a delegate instance with an appropriate set of arguments causes each of the delegate's callable entities to be invoked with the given set of arguments.

An interesting and useful property of a delegate instance is that it does not know or care about the classes of the methods it encapsulates; all that matters is that those methods be compatible ([Delegate declarations](#)) with the delegate's type. This makes delegates perfectly suited for "anonymous" invocation.

Delegate declarations

A *delegate_declaration* is a *type_declaration* ([Type declarations](#)) that declares a new delegate type.

```
delegate_declaration
: attributes? delegate_modifier* 'delegate' return_type
  identifier variant_type_parameter_list?
  '(' formal_parameter_list? ')' type_parameter_constraints_clause* ';'
;

delegate_modifier
: 'new'
| 'public'
| 'protected'
| 'internal'
| 'private'
| delegate_modifier_unsafe
;
```

It is a compile-time error for the same modifier to appear multiple times in a delegate declaration.

The `new` modifier is only permitted on delegates declared within another type, in which case it specifies that such a delegate hides an inherited member by the same name, as described in [The new modifier](#).

The `public`, `protected`, `internal`, and `private` modifiers control the accessibility of the delegate type. Depending on the context in which the delegate declaration occurs, some of these modifiers may not be permitted ([Declared accessibility](#)).

The delegate's type name is *identifier*.

The optional *formal_parameter_list* specifies the parameters of the delegate, and *return_type* indicates the return type of the delegate.

The optional *variant_type_parameter_list* ([Variant type parameter lists](#)) specifies the type parameters to the delegate itself.

The return type of a delegate type must be either `void`, or output-safe ([Variance safety](#)).

All the formal parameter types of a delegate type must be input-safe. Additionally, any `out` or `ref` parameter types must also be output-safe. Note that even `out` parameters are required to be input-safe, due to a limitation of the underlying execution platform.

Delegate types in C# are name equivalent, not structurally equivalent. Specifically, two different delegate types that have the same parameter lists and return type are considered different delegate types. However, instances of two distinct but structurally equivalent delegate types may compare as equal ([Delegate equality operators](#)).

For example:

```
delegate int D1(int i, double d);

class A
{
    public static int M1(int a, double b) {...}
}

class B
{
    delegate int D2(int c, double d);
    public static int M1(int f, double g) {...}
    public static void M2(int k, double l) {...}
    public static int M3(int g) {...}
    public static void M4(int g) {...}
}
```

The methods `A.M1` and `B.M1` are compatible with both the delegate types `D1` and `D2`, since they have the same return type and parameter list; however, these delegate types are two different types, so they are not interchangeable. The methods `B.M2`, `B.M3`, and `B.M4` are incompatible with the delegate types `D1` and `D2`, since they have different return types or parameter lists.

Like other generic type declarations, type arguments must be given to create a constructed delegate type. The parameter types and return type of a constructed delegate type are created by substituting, for each type parameter in the delegate declaration, the corresponding type argument of the constructed delegate type. The resulting return type and parameter types are used in determining what methods are compatible with a constructed delegate type. For example:

```
delegate bool Predicate<T>(T value);

class X
{
    static bool F(int i) {...}
    static bool G(string s) {...}
}
```

The method `X.F` is compatible with the delegate type `Predicate<int>` and the method `X.G` is compatible with the delegate type `Predicate<string>`.

The only way to declare a delegate type is via a *delegate declaration*. A delegate type is a class type that is derived from `System.Delegate`. Delegate types are implicitly `sealed`, so it is not permissible to derive any type from a delegate type. It is also not permissible to derive a non-delegate class type from `System.Delegate`. Note that `System.Delegate` is not itself a delegate type; it is a class type from which all delegate types are derived.

C# provides special syntax for delegate instantiation and invocation. Except for instantiation, any operation that can be applied to a class or class instance can also be applied to a delegate class or instance, respectively. In particular, it is possible to access members of the `System.Delegate` type via the usual member access syntax.

The set of methods encapsulated by a delegate instance is called an invocation list. When a delegate instance is

created ([Delegate compatibility](#)) from a single method, it encapsulates that method, and its invocation list contains only one entry. However, when two non-null delegate instances are combined, their invocation lists are concatenated -- in the order left operand then right operand -- to form a new invocation list, which contains two or more entries.

Delegates are combined using the binary `+` ([Addition operator](#)) and `+=` operators ([Compound assignment](#)). A delegate can be removed from a combination of delegates, using the binary `-` ([Subtraction operator](#)) and `-=` operators ([Compound assignment](#)). Delegates can be compared for equality ([Delegate equality operators](#)).

The following example shows the instantiation of a number of delegates, and their corresponding invocation lists:

```
delegate void D(int x);

class C
{
    public static void M1(int i) {...}
    public static void M2(int i) {...}
}

class Test
{
    static void Main() {
        D cd1 = new D(C.M1);           // M1
        D cd2 = new D(C.M2);           // M2
        D cd3 = cd1 + cd2;              // M1 + M2
        D cd4 = cd3 + cd1;              // M1 + M2 + M1
        D cd5 = cd4 + cd3;              // M1 + M2 + M1 + M1 + M2
    }
}
```

When `cd1` and `cd2` are instantiated, they each encapsulate one method. When `cd3` is instantiated, it has an invocation list of two methods, `M1` and `M2`, in that order. `cd4`'s invocation list contains `M1`, `M2`, and `M1`, in that order. Finally, `cd5`'s invocation list contains `M1`, `M2`, `M1`, `M1`, and `M2`, in that order. For more examples of combining (as well as removing) delegates, see [Delegate invocation](#).

Delegate compatibility

A method or delegate `M` is *compatible* with a delegate type `D` if all of the following are true:

- `D` and `M` have the same number of parameters, and each parameter in `D` has the same `ref` or `out` modifiers as the corresponding parameter in `M`.
- For each value parameter (a parameter with no `ref` or `out` modifier), an identity conversion ([Identity conversion](#)) or implicit reference conversion ([Implicit reference conversions](#)) exists from the parameter type in `D` to the corresponding parameter type in `M`.
- For each `ref` or `out` parameter, the parameter type in `D` is the same as the parameter type in `M`.
- An identity or implicit reference conversion exists from the return type of `M` to the return type of `D`.

Delegate instantiation

An instance of a delegate is created by a *delegate_creation_expression* ([Delegate creation expressions](#)) or a conversion to a delegate type. The newly created delegate instance then refers to either:

- The static method referenced in the *delegate_creation_expression*, or
- The target object (which cannot be `null`) and instance method referenced in the

delegate_creation_expression, or

- Another delegate.

For example:

```
delegate void D(int x);

class C
{
    public static void M1(int i) {...}
    public void M2(int i) {...}
}

class Test
{
    static void Main() {
        D cd1 = new D(C.M1);           // static method
        C t = new C();
        D cd2 = new D(t.M2);           // instance method
        D cd3 = new D(cd2);            // another delegate
    }
}
```

Once instantiated, delegate instances always refer to the same target object and method. Remember, when two delegates are combined, or one is removed from another, a new delegate results with its own invocation list; the invocation lists of the delegates combined or removed remain unchanged.

Delegate invocation

C# provides special syntax for invoking a delegate. When a non-null delegate instance whose invocation list contains one entry is invoked, it invokes the one method with the same arguments it was given, and returns the same value as the referred to method. (See [Delegate invocations](#) for detailed information on delegate invocation.) If an exception occurs during the invocation of such a delegate, and that exception is not caught within the method that was invoked, the search for an exception catch clause continues in the method that called the delegate, as if that method had directly called the method to which that delegate referred.

Invocation of a delegate instance whose invocation list contains multiple entries proceeds by invoking each of the methods in the invocation list, synchronously, in order. Each method so called is passed the same set of arguments as was given to the delegate instance. If such a delegate invocation includes reference parameters ([Reference parameters](#)), each method invocation will occur with a reference to the same variable; changes to that variable by one method in the invocation list will be visible to methods further down the invocation list. If the delegate invocation includes output parameters or a return value, their final value will come from the invocation of the last delegate in the list.

If an exception occurs during processing of the invocation of such a delegate, and that exception is not caught within the method that was invoked, the search for an exception catch clause continues in the method that called the delegate, and any methods further down the invocation list are not invoked.

Attempting to invoke a delegate instance whose value is null results in an exception of type

```
System.NullReferenceException .
```

The following example shows how to instantiate, combine, remove, and invoke delegates:


```

using System;

delegate void D(int x);

class C
{
    public static void M1(int i) {
        Console.WriteLine("C.M1: " + i);
    }

    public static void M2(int i) {
        Console.WriteLine("C.M2: " + i);
    }

    public void M3(int i) {
        Console.WriteLine("C.M3: " + i);
    }
}

class Test
{
    static void Main() {
        D cd1 = new D(C.M1);
        cd1(-1);           // call M1

        D cd2 = new D(C.M2);
        cd2(-2);           // call M2

        D cd3 = cd1 + cd2;
        cd3(10);           // call M1 then M2

        cd3 += cd1;
        cd3(20);           // call M1, M2, then M1

        C c = new C();
        D cd4 = new D(c.M3);
        cd3 += cd4;
        cd3(30);           // call M1, M2, M1, then M3

        cd3 -= cd1;         // remove last M1
        cd3(40);           // call M1, M2, then M3

        cd3 -= cd4;
        cd3(50);           // call M1 then M2

        cd3 -= cd2;
        cd3(60);           // call M1

        cd3 -= cd2;         // impossible removal is benign
        cd3(60);           // call M1

        cd3 -= cd1;         // invocation list is empty so cd3 is null

        cd3(70);           // System.NullReferenceException thrown

        cd3 -= cd1;         // impossible removal is benign
    }
}

```

As shown in the statement `cd3 += cd1;`, a delegate can be present in an invocation list multiple times. In this case, it is simply invoked once per occurrence. In an invocation list such as this, when that delegate is removed, the last occurrence in the invocation list is the one actually removed.

Immediately prior to the execution of the final statement, `cd3 -= cd1;`, the delegate `cd3` refers to an empty invocation list. Attempting to remove a delegate from an empty list (or to remove a non-existent delegate from

a non-empty list) is not an error.

The output produced is:

```
C.M1: -1  
C.M2: -2  
C.M1: 10  
C.M2: 10  
C.M1: 20  
C.M2: 20  
C.M1: 20  
C.M1: 30  
C.M2: 30  
C.M1: 30  
C.M3: 30  
C.M1: 40  
C.M2: 40  
C.M3: 40  
C.M1: 50  
C.M2: 50  
C.M1: 60  
C.M1: 60
```

Exceptions

12/28/2021 • 4 minutes to read • [Edit Online](#)

Exceptions in C# provide a structured, uniform, and type-safe way of handling both system level and application level error conditions. The exception mechanism in C# is quite similar to that of C++, with a few important differences:

- In C#, all exceptions must be represented by an instance of a class type derived from `System.Exception`. In C++, any value of any type can be used to represent an exception.
- In C#, a finally block ([The try statement](#)) can be used to write termination code that executes in both normal execution and exceptional conditions. Such code is difficult to write in C++ without duplicating code.
- In C#, system-level exceptions such as overflow, divide-by-zero, and null dereferences have well defined exception classes and are on a par with application-level error conditions.

Causes of exceptions

Exception can be thrown in two different ways.

- A `throw` statement ([The throw statement](#)) throws an exception immediately and unconditionally. Control never reaches the statement immediately following the `throw`.
- Certain exceptional conditions that arise during the processing of C# statements and expression cause an exception in certain circumstances when the operation cannot be completed normally. For example, an integer division operation ([Division operator](#)) throws a `System.DivideByZeroException` if the denominator is zero. See [Common Exception Classes](#) for a list of the various exceptions that can occur in this way.

The System.Exception class

The `System.Exception` class is the base type of all exceptions. This class has a few notable properties that all exceptions share:

- `Message` is a read-only property of type `string` that contains a human-readable description of the reason for the exception.
- `InnerException` is a read-only property of type `Exception`. If its value is non-null, it refers to the exception that caused the current exception—that is, the current exception was raised in a catch block handling the `InnerException`. Otherwise, its value is null, indicating that this exception was not caused by another exception. The number of exception objects chained together in this manner can be arbitrary.

The value of these properties can be specified in calls to the instance constructor for `System.Exception`.

How exceptions are handled

Exceptions are handled by a `try` statement ([The try statement](#)).

When an exception occurs, the system searches for the nearest `catch` clause that can handle the exception, as determined by the run-time type of the exception. First, the current method is searched for a lexically enclosing `try` statement, and the associated catch clauses of the try statement are considered in order. If that fails, the method that called the current method is searched for a lexically enclosing `try` statement that encloses the point of the call to the current method. This search continues until a `catch` clause is found that can handle the current exception, by naming an exception class that is of the same class, or a base class, of the run-time type of the exception being thrown. A `catch` clause that doesn't name an exception class can handle any exception.

Once a matching catch clause is found, the system prepares to transfer control to the first statement of the catch clause. Before execution of the catch clause begins, the system first executes, in order, any `finally` clauses that were associated with try statements more nested than the one that caught the exception.

If no matching catch clause is found, one of two things occurs:

- If the search for a matching catch clause reaches a static constructor ([Static constructors](#)) or static field initializer, then a `System.TypeInitializationException` is thrown at the point that triggered the invocation of the static constructor. The inner exception of the `System.TypeInitializationException` contains the exception that was originally thrown.
- If the search for matching catch clauses reaches the code that initially started the thread, then execution of the thread is terminated. The impact of such termination is implementation-defined.

Exceptions that occur during destructor execution are worth special mention. If an exception occurs during destructor execution, and that exception is not caught, then the execution of that destructor is terminated and the destructor of the base class (if any) is called. If there is no base class (as in the case of the `object` type) or if there is no base class destructor, then the exception is discarded.

Common Exception Classes

The following exceptions are thrown by certain C# operations.

EXCEPTION TYPE	DESCRIPTION
<code>System.ArithmeticException</code>	A base class for exceptions that occur during arithmetic operations, such as <code>System.DivideByZeroException</code> and <code>System.OverflowException</code> .
<code>System.ArrayTypeMismatchException</code>	Thrown when a store into an array fails because the actual type of the stored element is incompatible with the actual type of the array.
<code>System.DivideByZeroException</code>	Thrown when an attempt to divide an integral value by zero occurs.
<code>System.IndexOutOfRangeException</code>	Thrown when an attempt to index an array via an index that is less than zero or outside the bounds of the array.
<code>System.InvalidCastException</code>	Thrown when an explicit conversion from a base type or interface to a derived type fails at run time.
<code>System.NullReferenceException</code>	Thrown when a <code>null</code> reference is used in a way that causes the referenced object to be required.
<code>System.OutOfMemoryException</code>	Thrown when an attempt to allocate memory (via <code>new</code>) fails.
<code>System.OverflowException</code>	Thrown when an arithmetic operation in a <code>checked</code> context overflows.
<code>System.StackOverflowException</code>	Thrown when the execution stack is exhausted by having too many pending method calls; typically indicative of very deep or unbounded recursion.

EXCEPTION TYPE	DESCRIPTION
<code>System.TypeInitializationException</code>	Thrown when a static constructor throws an exception, and no <code>catch</code> clauses exists to catch it.

Attributes

12/28/2021 • 25 minutes to read • [Edit Online](#)

Much of the C# language enables the programmer to specify declarative information about the entities defined in the program. For example, the accessibility of a method in a class is specified by decorating it with the *method_modifiers* `public`, `protected`, `internal`, and `private`.

C# enables programmers to invent new kinds of declarative information, called **attributes**. Programmers can then attach attributes to various program entities, and retrieve attribute information in a run-time environment. For instance, a framework might define a `HelpAttribute` attribute that can be placed on certain program elements (such as classes and methods) to provide a mapping from those program elements to their documentation.

Attributes are defined through the declaration of attribute classes ([Attribute classes](#)), which may have positional and named parameters ([Positional and named parameters](#)). Attributes are attached to entities in a C# program using attribute specifications ([Attribute specification](#)), and can be retrieved at run-time as attribute instances ([Attribute instances](#)).

Attribute classes

A class that derives from the abstract class `System.Attribute`, whether directly or indirectly, is an **attribute class**. The declaration of an attribute class defines a new kind of **attribute** that can be placed on a declaration. By convention, attribute classes are named with a suffix of `Attribute`. Uses of an attribute may either include or omit this suffix.

Attribute usage

The attribute `AttributeUsage` ([The AttributeUsage attribute](#)) is used to describe how an attribute class can be used.

`AttributeUsage` has a positional parameter ([Positional and named parameters](#)) that enables an attribute class to specify the kinds of declarations on which it can be used. The example

```
using System;

[AttributeUsage(AttributeTargets.Class | AttributeTargets.Interface)]
public class SimpleAttribute: Attribute
{
    ...
}
```

defines an attribute class named `SimpleAttribute` that can be placed on *class_declarations* and *interface_declarations* only. The example

```
[Simple] class Class1 {...}

[Simple] interface Interface1 {...}
```

shows several uses of the `Simple` attribute. Although this attribute is defined with the name `SimpleAttribute`, when this attribute is used, the `Attribute` suffix may be omitted, resulting in the short name `Simple`. Thus, the example above is semantically equivalent to the following:

```
[SimpleAttribute] class Class1 {...}

[SimpleAttribute] interface Interface1 {...}
```

`AttributeUsage` has a named parameter ([Positional and named parameters](#)) called `AllowMultiple`, which indicates whether the attribute can be specified more than once for a given entity. If `AllowMultiple` for an attribute class is true, then that attribute class is a *multi-use attribute class*, and can be specified more than once on an entity. If `AllowMultiple` for an attribute class is false or it is unspecified, then that attribute class is a *single-use attribute class*, and can be specified at most once on an entity.

The example

```
using System;

[AttributeUsage(AttributeTargets.Class, AllowMultiple = true)]
public class AuthorAttribute: Attribute
{
    private string name;

    public AuthorAttribute(string name) {
        this.name = name;
    }

    public string Name {
        get { return name; }
    }
}
```

defines a multi-use attribute class named `AuthorAttribute`. The example

```
[Author("Brian Kernighan"), Author("Dennis Ritchie")]
class Class1
{
    ...
}
```

shows a class declaration with two uses of the `Author` attribute.

`AttributeUsage` has another named parameter called `Inherited`, which indicates whether the attribute, when specified on a base class, is also inherited by classes that derive from that base class. If `Inherited` for an attribute class is true, then that attribute is inherited. If `Inherited` for an attribute class is false then that attribute is not inherited. If it is unspecified, its default value is true.

An attribute class `X` not having an `AttributeUsage` attribute attached to it, as in

```
using System;

class X: Attribute {...}
```

is equivalent to the following:

```
using System;

[AttributeUsage(
    AttributeTargets.All,
    AllowMultiple = false,
    Inherited = true)
]
class X: Attribute {...}
```

Positional and named parameters

Attribute classes can have *positional parameters* and *named parameters*. Each public instance constructor for an attribute class defines a valid sequence of positional parameters for that attribute class. Each non-static public read-write field and property for an attribute class defines a named parameter for the attribute class.

The example

```
using System;

[AttributeUsage(AttributeTargets.Class)]
public class HelpAttribute: Attribute
{
    public HelpAttribute(string url) {           // Positional parameter
        ...
    }

    public string Topic {                       // Named parameter
        get {...}
        set {...}
    }

    public string Url {
        get {...}
    }
}
```

defines an attribute class named `HelpAttribute` that has one positional parameter, `url`, and one named parameter, `Topic`. Although it is non-static and public, the property `url` does not define a named parameter, since it is not read-write.

This attribute class might be used as follows:

```
[Help("http://www.mycompany.com/.../Class1.htm")]
class Class1
{
    ...
}

[Help("http://www.mycompany.com/.../Misc.htm", Topic = "Class2")]
class Class2
{
    ...
}
```

Attribute parameter types

The types of positional and named parameters for an attribute class are limited to the *attribute parameter types*, which are:

- One of the following types: `bool`, `byte`, `char`, `double`, `float`, `int`, `long`, `sbyte`, `short`, `string`, `uint`, `ulong`, `ushort`.

- The type `object` .
- The type `System.Type` .
- An enum type, provided it has public accessibility and the types in which it is nested (if any) also have public accessibility ([Attribute specification](#)).
- Single-dimensional arrays of the above types.
- A constructor argument or public field which does not have one of these types, cannot be used as a positional or named parameter in an attribute specification.

Attribute specification

Attribute specification is the application of a previously defined attribute to a declaration. An attribute is a piece of additional declarative information that is specified for a declaration. Attributes can be specified at global scope (to specify attributes on the containing assembly or module) and for *type_declarations* ([Type declarations](#)), *class_member_declarations* ([Type parameter constraints](#)), *interface_member_declarations* ([Interface members](#)), *struct_member_declarations* ([Struct members](#)), *enum_member_declarations* ([Enum members](#)), *accessor_declarations* ([Accessors](#)), *event_accessor_declarations* ([Field-like events](#)), and *formal_parameter_lists* ([Method parameters](#)).

Attributes are specified in **attribute sections**. An attribute section consists of a pair of square brackets, which surround a comma-separated list of one or more attributes. The order in which attributes are specified in such a list, and the order in which sections attached to the same program entity are arranged, is not significant. For instance, the attribute specifications `[A][B]` , `[B][A]` , `[A,B]` , and `[B,A]` are equivalent.

```
global_attributes
: global_attribute_section+
;

global_attribute_section
: '[' global_attribute_target_specifier attribute_list ']'
| '[' global_attribute_target_specifier attribute_list ',' ']'
;

global_attribute_target_specifier
: global_attribute_target ':'
;

global_attribute_target
: 'assembly'
| 'module'
;

attributes
: attribute_section+
;

attribute_section
: '[' attribute_target_specifier? attribute_list ']'
| '[' attribute_target_specifier? attribute_list ',' ']'
;

attribute_target_specifier
: attribute_target ':'
;

attribute_target
: 'field'
| 'event'
| 'method'
| 'param'
| 'property'
| 'return'
```

```

    | 'type'
    ;

attribute_list
    : attribute (',' attribute)*
    ;

attribute
    : attribute_name attribute_arguments?
    ;

attribute_name
    : type_name
    ;

attribute_arguments
    : '(' positional_argument_list? ')'
    | '(' positional_argument_list ',' named_argument_list ')'
    | '(' named_argument_list ')'
    ;

positional_argument_list
    : positional_argument (',' positional_argument)*
    ;

positional_argument
    : attribute_argument_expression
    ;

named_argument_list
    : named_argument (',' named_argument)*
    ;

named_argument
    : identifier '=' attribute_argument_expression
    ;

attribute_argument_expression
    : expression
    ;

```

An attribute consists of an *attribute_name* and an optional list of positional and named arguments. The positional arguments (if any) precede the named arguments. A positional argument consists of an *attribute_argument_expression*; a named argument consists of a name, followed by an equal sign, followed by an *attribute_argument_expression*, which, together, are constrained by the same rules as simple assignment. The order of named arguments is not significant.

The *attribute_name* identifies an attribute class. If the form of *attribute_name* is *type_name* then this name must refer to an attribute class. Otherwise, a compile-time error occurs. The example

```

class Class1 {}

[Class1] class Class2 {}    // Error

```

results in a compile-time error because it attempts to use `Class1` as an attribute class when `Class1` is not an attribute class.

Certain contexts permit the specification of an attribute on more than one target. A program can explicitly specify the target by including an *attribute_target_specifier*. When an attribute is placed at the global level, a *global_attribute_target_specifier* is required. In all other locations, a reasonable default is applied, but an *attribute_target_specifier* can be used to affirm or override the default in certain ambiguous cases (or to just affirm the default in non-ambiguous cases). Thus, typically, *attribute_target_specifiers* can be omitted except at

the global level. The potentially ambiguous contexts are resolved as follows:

- An attribute specified at global scope can apply either to the target assembly or the target module. No default exists for this context, so an *attribute_target_specifier* is always required in this context. The presence of the `assembly` *attribute_target_specifier* indicates that the attribute applies to the target assembly; the presence of the `module` *attribute_target_specifier* indicates that the attribute applies to the target module.
- An attribute specified on a delegate declaration can apply either to the delegate being declared or to its return value. In the absence of an *attribute_target_specifier*, the attribute applies to the delegate. The presence of the `type` *attribute_target_specifier* indicates that the attribute applies to the delegate; the presence of the `return` *attribute_target_specifier* indicates that the attribute applies to the return value.
- An attribute specified on a method declaration can apply either to the method being declared or to its return value. In the absence of an *attribute_target_specifier*, the attribute applies to the method. The presence of the `method` *attribute_target_specifier* indicates that the attribute applies to the method; the presence of the `return` *attribute_target_specifier* indicates that the attribute applies to the return value.
- An attribute specified on an operator declaration can apply either to the operator being declared or to its return value. In the absence of an *attribute_target_specifier*, the attribute applies to the operator. The presence of the `method` *attribute_target_specifier* indicates that the attribute applies to the operator; the presence of the `return` *attribute_target_specifier* indicates that the attribute applies to the return value.
- An attribute specified on an event declaration that omits event accessors can apply to the event being declared, to the associated field (if the event is not abstract), or to the associated add and remove methods. In the absence of an *attribute_target_specifier*, the attribute applies to the event. The presence of the `event` *attribute_target_specifier* indicates that the attribute applies to the event; the presence of the `field` *attribute_target_specifier* indicates that the attribute applies to the field; and the presence of the `method` *attribute_target_specifier* indicates that the attribute applies to the methods.
- An attribute specified on a get accessor declaration for a property or indexer declaration can apply either to the associated method or to its return value. In the absence of an *attribute_target_specifier*, the attribute applies to the method. The presence of the `method` *attribute_target_specifier* indicates that the attribute applies to the method; the presence of the `return` *attribute_target_specifier* indicates that the attribute applies to the return value.
- An attribute specified on a set accessor for a property or indexer declaration can apply either to the associated method or to its lone implicit parameter. In the absence of an *attribute_target_specifier*, the attribute applies to the method. The presence of the `method` *attribute_target_specifier* indicates that the attribute applies to the method; the presence of the `param` *attribute_target_specifier* indicates that the attribute applies to the parameter; the presence of the `return` *attribute_target_specifier* indicates that the attribute applies to the return value.
- An attribute specified on an add or remove accessor declaration for an event declaration can apply either to the associated method or to its lone parameter. In the absence of an *attribute_target_specifier*, the attribute applies to the method. The presence of the `method` *attribute_target_specifier* indicates that the attribute applies to the method; the presence of the `param` *attribute_target_specifier* indicates that the attribute applies to the parameter; the presence of the `return` *attribute_target_specifier* indicates that the attribute applies to the return value.

In other contexts, inclusion of an *attribute_target_specifier* is permitted but unnecessary. For instance, a class declaration may either include or omit the specifier `type`:

```
[type: Author("Brian Kernighan")]
class Class1 {}

[Author("Dennis Ritchie")]
class Class2 {}
```

It is an error to specify an invalid *attribute_target_specifier*. For instance, the specifier `param` cannot be used on

a class declaration:

```
[param: Author("Brian Kernighan")]      // Error
class Class1 {}
```

By convention, attribute classes are named with a suffix of `Attribute`. An *attribute_name* of the form *type_name* may either include or omit this suffix. If an attribute class is found both with and without this suffix, an ambiguity is present, and a compile-time error results. If the *attribute_name* is spelled such that its right-most *identifier* is a verbatim identifier ([Identifiers](#)), then only an attribute without a suffix is matched, thus enabling such an ambiguity to be resolved. The example

```
using System;

[AttributeUsage(AttributeTargets.All)]
public class X: Attribute
{}

[AttributeUsage(AttributeTargets.All)]
public class XAttribute: Attribute
{}

[X]                                // Error: ambiguity
class Class1 {}

[XAttribute]                       // Refers to XAttribute
class Class2 {}

[@X]                              // Refers to X
class Class3 {}

[@XAttribute]                     // Refers to XAttribute
class Class4 {}
```

shows two attribute classes named `X` and `XAttribute`. The attribute `[X]` is ambiguous, since it could refer to either `X` or `XAttribute`. Using a verbatim identifier allows the exact intent to be specified in such rare cases. The attribute `[XAttribute]` is not ambiguous (although it would be if there was an attribute class named `XAttributeAttribute`!). If the declaration for class `X` is removed, then both attributes refer to the attribute class named `XAttribute`, as follows:

```
using System;

[AttributeUsage(AttributeTargets.All)]
public class XAttribute: Attribute
{}

[X]                                // Refers to XAttribute
class Class1 {}

[XAttribute]                       // Refers to XAttribute
class Class2 {}

[@X]                              // Error: no attribute named "X"
class Class3 {}
```

It is a compile-time error to use a single-use attribute class more than once on the same entity. The example

```

using System;

[AttributeUsage(AttributeTargets.Class)]
public class HelpStringAttribute: Attribute
{
    string value;

    public HelpStringAttribute(string value) {
        this.value = value;
    }

    public string Value {
        get {...}
    }
}

[HelpString("Description of Class1")]
[HelpString("Another description of Class1")]
public class Class1 {}

```

results in a compile-time error because it attempts to use `HelpString`, which is a single-use attribute class, more than once on the declaration of `Class1`.

An expression `E` is an *attribute_argument_expression* if all of the following statements are true:

- The type of `E` is an attribute parameter type ([Attribute parameter types](#)).
- At compile-time, the value of `E` can be resolved to one of the following:
 - A constant value.
 - A `System.Type` object.
 - A one-dimensional array of *attribute_argument_expressions*.

For example:

```

using System;

[AttributeUsage(AttributeTargets.Class)]
public class TestAttribute: Attribute
{
    public int P1 {
        get {...}
        set {...}
    }

    public Type P2 {
        get {...}
        set {...}
    }

    public object P3 {
        get {...}
        set {...}
    }
}

[Test(P1 = 1234, P3 = new int[] {1, 3, 5}, P2 = typeof(float))]
class MyClass {}

```

A *typeof_expression* ([The typeof operator](#)) used as an attribute argument expression can reference a non-generic type, a closed constructed type, or an unbound generic type, but it cannot reference an open type. This is to ensure that the expression can be resolved at compile-time.

```

class A: Attribute
{
    public A(Type t) {...}
}

class G<T>
{
    [A(typeof(T))] T t;           // Error, open type in attribute
}

class X
{
    [A(typeof(List<int>))] int x;   // Ok, closed constructed type
    [A(typeof(List<>))] int y;     // Ok, unbound generic type
}

```

Attribute instances

An **attribute instance** is an instance that represents an attribute at run-time. An attribute is defined with an attribute class, positional arguments, and named arguments. An attribute instance is an instance of the attribute class that is initialized with the positional and named arguments.

Retrieval of an attribute instance involves both compile-time and run-time processing, as described in the following sections.

Compilation of an attribute

The compilation of an *attribute* with attribute class `T`, *positional_argument_list* `P` and *named_argument_list* `N`, consists of the following steps:

- Follow the compile-time processing steps for compiling an *object_creation_expression* of the form `new T(P)`. These steps either result in a compile-time error, or determine an instance constructor `C` on `T` that can be invoked at run-time.
- If `C` does not have public accessibility, then a compile-time error occurs.
- For each *named_argument* `Arg` in `N`:
 - Let `Name` be the *identifier* of the *named_argument* `Arg`.
 - `Name` must identify a non-static read-write public field or property on `T`. If `T` has no such field or property, then a compile-time error occurs.
- Keep the following information for run-time instantiation of the attribute: the attribute class `T`, the instance constructor `C` on `T`, the *positional_argument_list* `P` and the *named_argument_list* `N`.

Run-time retrieval of an attribute instance

Compilation of an *attribute* yields an attribute class `T`, an instance constructor `C` on `T`, a *positional_argument_list* `P`, and a *named_argument_list* `N`. Given this information, an attribute instance can be retrieved at run-time using the following steps:

- Follow the run-time processing steps for executing an *object_creation_expression* of the form `new T(P)`, using the instance constructor `C` as determined at compile-time. These steps either result in an exception, or produce an instance `O` of `T`.
- For each *named_argument* `Arg` in `N`, in order:
 - Let `Name` be the *identifier* of the *named_argument* `Arg`. If `Name` does not identify a non-static public read-write field or property on `O`, then an exception is thrown.
 - Let `Value` be the result of evaluating the *attribute_argument_expression* of `Arg`.
 - If `Name` identifies a field on `O`, then set this field to `Value`.
 - Otherwise, `Name` identifies a property on `O`. Set this property to `Value`.

- The result is `o`, an instance of the attribute class `T` that has been initialized with the *positional_argument_list* `P` and the *named_argument_list* `N`.

Reserved attributes

A small number of attributes affect the language in some way. These attributes include:

- `System.AttributeUsageAttribute` ([The AttributeUsage attribute](#)), which is used to describe the ways in which an attribute class can be used.
- `System.Diagnostics.ConditionalAttribute` ([The Conditional attribute](#)), which is used to define conditional methods.
- `System.ObsoleteAttribute` ([The Obsolete attribute](#)), which is used to mark a member as obsolete.
- `System.Runtime.CompilerServices.CallerLineNumberAttribute`, `System.Runtime.CompilerServices.CallerFilePathAttribute` and `System.Runtime.CompilerServices.CallerMemberNameAttribute` ([Caller info attributes](#)), which are used to supply information about the calling context to optional parameters.

The AttributeUsage attribute

The attribute `AttributeUsage` is used to describe the manner in which the attribute class can be used.

A class that is decorated with the `AttributeUsage` attribute must derive from `System.Attribute`, either directly or indirectly. Otherwise, a compile-time error occurs.

```
namespace System
{
    [AttributeUsage(AttributeTargets.Class)]
    public class AttributeUsageAttribute : Attribute
    {
        public AttributeUsageAttribute(AttributeTargets validOn) {...}
        public virtual bool AllowMultiple { get {...} set {...} }
        public virtual bool Inherited { get {...} set {...} }
        public virtual AttributeTargets ValidOn { get {...} }
    }

    public enum AttributeTargets
    {
        Assembly      = 0x0001,
        Module        = 0x0002,
        Class          = 0x0004,
        Struct         = 0x0008,
        Enum           = 0x0010,
        Constructor    = 0x0020,
        Method         = 0x0040,
        Property       = 0x0080,
        Field          = 0x0100,
        Event          = 0x0200,
        Interface      = 0x0400,
        Parameter      = 0x0800,
        Delegate       = 0x1000,
        ReturnValue    = 0x2000,

        All = Assembly | Module | Class | Struct | Enum | Constructor |
            Method | Property | Field | Event | Interface | Parameter |
            Delegate | ReturnValue
    }
}
```

The Conditional attribute

The attribute `Conditional` enables the definition of *conditional methods* and *conditional attribute classes*.

```

namespace System.Diagnostics
{
    [AttributeUsage(AttributeTargets.Method | AttributeTargets.Class, AllowMultiple = true)]
    public class ConditionalAttribute: Attribute
    {
        public ConditionalAttribute(string conditionString) {...}
        public string ConditionString { get {...} }
    }
}

```

Conditional methods

A method decorated with the `Conditional` attribute is a conditional method. The `Conditional` attribute indicates a condition by testing a conditional compilation symbol. Calls to a conditional method are either included or omitted depending on whether this symbol is defined at the point of the call. If the symbol is defined, the call is included; otherwise, the call (including evaluation of the receiver and parameters of the call) is omitted.

A conditional method is subject to the following restrictions:

- The conditional method must be a method in a *class_declaration* or *struct_declaration*. A compile-time error occurs if the `Conditional` attribute is specified on a method in an interface declaration.
- The conditional method must have a return type of `void`.
- The conditional method must not be marked with the `override` modifier. A conditional method may be marked with the `virtual` modifier, however. Overrides of such a method are implicitly conditional, and must not be explicitly marked with a `Conditional` attribute.
- The conditional method must not be an implementation of an interface method. Otherwise, a compile-time error occurs.

In addition, a compile-time error occurs if a conditional method is used in a *delegate_creation_expression*. The example

```

#define DEBUG

using System;
using System.Diagnostics;

class Class1
{
    [Conditional("DEBUG")]
    public static void M() {
        Console.WriteLine("Executed Class1.M");
    }
}

class Class2
{
    public static void Test() {
        Class1.M();
    }
}

```

declares `Class1.M` as a conditional method. `Class2`'s `Test` method calls this method. Since the conditional compilation symbol `DEBUG` is defined, if `Class2.Test` is called, it will call `M`. If the symbol `DEBUG` had not been defined, then `Class2.Test` would not call `Class1.M`.

It is important to note that the inclusion or exclusion of a call to a conditional method is controlled by the conditional compilation symbols at the point of the call. In the example

File `class1.cs` :

```
using System.Diagnostics;

class Class1
{
    [Conditional("DEBUG")]
    public static void F() {
        Console.WriteLine("Executed Class1.F");
    }
}
```

File `class2.cs` :

```
#define DEBUG

class Class2
{
    public static void G() {
        Class1.F();           // F is called
    }
}
```

File `class3.cs` :

```
#undef DEBUG

class Class3
{
    public static void H() {
        Class1.F();           // F is not called
    }
}
```

the classes `Class2` and `Class3` each contain calls to the conditional method `Class1.F`, which is conditional based on whether or not `DEBUG` is defined. Since this symbol is defined in the context of `Class2` but not `Class3`, the call to `F` in `Class2` is included, while the call to `F` in `Class3` is omitted.

The use of conditional methods in an inheritance chain can be confusing. Calls made to a conditional method through `base`, of the form `base.M`, are subject to the normal conditional method call rules. In the example

File `class1.cs` :

```
using System;
using System.Diagnostics;

class Class1
{
    [Conditional("DEBUG")]
    public virtual void M() {
        Console.WriteLine("Class1.M executed");
    }
}
```

File `class2.cs` :

```
using System;

class Class2: Class1
{
    public override void M() {
        Console.WriteLine("Class2.M executed");
        base.M();                // base.M is not called!
    }
}
```

File `class3.cs` :

```
#define DEBUG

using System;

class Class3
{
    public static void Test() {
        Class2 c = new Class2();
        c.M();                // M is called
    }
}
```

`Class2` includes a call to the `M` defined in its base class. This call is omitted because the base method is conditional based on the presence of the symbol `DEBUG`, which is undefined. Thus, the method writes to the console "`Class2.M executed`" only. Judicious use of *pp_declarations* can eliminate such problems.

Conditional attribute classes

An attribute class ([Attribute classes](#)) decorated with one or more `Conditional` attributes is a ***conditional attribute class***. A conditional attribute class is thus associated with the conditional compilation symbols declared in its `Conditional` attributes. This example:

```
using System;
using System.Diagnostics;
[Conditional("ALPHA")]
[Conditional("BETA")]
public class TestAttribute : Attribute {}
```

declares `TestAttribute` as a conditional attribute class associated with the conditional compilations symbols `ALPHA` and `BETA`.

Attribute specifications ([Attribute specification](#)) of a conditional attribute are included if one or more of its associated conditional compilation symbols is defined at the point of specification, otherwise the attribute specification is omitted.

It is important to note that the inclusion or exclusion of an attribute specification of a conditional attribute class is controlled by the conditional compilation symbols at the point of the specification. In the example

File `test.cs` :

```
using System;
using System.Diagnostics;

[Conditional("DEBUG")]

public class TestAttribute : Attribute {}
```

File `class1.cs` :

```
#define DEBUG

[Test]           // TestAttribute is specified

class Class1 {}
```

File `class2.cs` :

```
#undef DEBUG

[Test]           // TestAttribute is not specified

class Class2 {}
```

the classes `Class1` and `Class2` are each decorated with attribute `Test`, which is conditional based on whether or not `DEBUG` is defined. Since this symbol is defined in the context of `Class1` but not `Class2`, the specification of the `Test` attribute on `Class1` is included, while the specification of the `Test` attribute on `Class2` is omitted.

The Obsolete attribute

The attribute `Obsolete` is used to mark types and members of types that should no longer be used.

```
namespace System
{
    [AttributeUsage(
        AttributeTargets.Class |
        AttributeTargets.Struct |
        AttributeTargets.Enum |
        AttributeTargets.Interface |
        AttributeTargets.Delegate |
        AttributeTargets.Method |
        AttributeTargets.Constructor |
        AttributeTargets.Property |
        AttributeTargets.Field |
        AttributeTargets.Event,
        Inherited = false)
    ]
    public class ObsoleteAttribute: Attribute
    {
        public ObsoleteAttribute() {...}
        public ObsoleteAttribute(string message) {...}
        public ObsoleteAttribute(string message, bool error) {...}
        public string Message { get {...} }
        public bool IsError { get {...} }
    }
}
```

If a program uses a type or member that is decorated with the `Obsolete` attribute, the compiler issues a warning or an error. Specifically, the compiler issues a warning if no error parameter is provided, or if the error parameter is provided and has the value `false`. The compiler issues an error if the error parameter is specified and has the value `true`.

In the example

```
[Obsolete("This class is obsolete; use class B instead")]
class A
{
    public void F() {}
}

class B
{
    public void F() {}
}

class Test
{
    static void Main() {
        A a = new A();          // Warning
        a.F();
    }
}
```

the class `A` is decorated with the `Obsolete` attribute. Each use of `A` in `Main` results in a warning that includes the specified message, "This class is obsolete; use class B instead."

Caller info attributes

For purposes such as logging and reporting, it is sometimes useful for a function member to obtain certain compile-time information about the calling code. The caller info attributes provide a way to pass such information transparently.

When an optional parameter is annotated with one of the caller info attributes, omitting the corresponding argument in a call does not necessarily cause the default parameter value to be substituted. Instead, if the specified information about the calling context is available, that information will be passed as the argument value.

For example:

```
using System.Runtime.CompilerServices

...

public void Log(
    [CallerLineNumber] int line = -1,
    [CallerFilePath] string path = null,
    [CallerMemberName] string name = null
)
{
    Console.WriteLine((line < 0) ? "No line" : "Line " + line);
    Console.WriteLine((path == null) ? "No file path" : path);
    Console.WriteLine((name == null) ? "No member name" : name);
}
```

A call to `Log()` with no arguments would print the line number and file path of the call, as well as the name of the member within which the call occurred.

Caller info attributes can occur on optional parameters anywhere, including in delegate declarations. However, the specific caller info attributes have restrictions on the types of the parameters they can attribute, so that there will always be an implicit conversion from a substituted value to the parameter type.

It is an error to have the same caller info attribute on a parameter of both the defining and implementing part of a partial method declaration. Only caller info attributes in the defining part are applied, whereas caller info attributes occurring only in the implementing part are ignored.

Caller information does not affect overload resolution. As the attributed optional parameters are still omitted from the source code of the caller, overload resolution ignores those parameters in the same way it ignores other omitted optional parameters ([Overload resolution](#)).

Caller information is only substituted when a function is explicitly invoked in source code. Implicit invocations such as implicit parent constructor calls do not have a source location and will not substitute caller information. Also, calls that are dynamically bound will not substitute caller information. When a caller info attributed parameter is omitted in such cases, the specified default value of the parameter is used instead.

One exception is query-expressions. These are considered syntactic expansions, and if the calls they expand to omit optional parameters with caller info attributes, caller information will be substituted. The location used is the location of the query clause which the call was generated from.

If more than one caller info attribute is specified on a given parameter, they are preferred in the following order:

`CallerLineNumber`, `CallerFilePath`, `CallerMemberName`.

The `CallerLineNumber` attribute

The `System.Runtime.CompilerServices.CallerLineNumberAttribute` is allowed on optional parameters when there is a standard implicit conversion ([Standard implicit conversions](#)) from the constant value `int.MaxValue` to the parameter's type. This ensures that any non-negative line number up to that value can be passed without error.

If a function invocation from a location in source code omits an optional parameter with the `CallerLineNumberAttribute`, then a numeric literal representing that location's line number is used as an argument to the invocation instead of the default parameter value.

If the invocation spans multiple lines, the line chosen is implementation-dependent.

Note that the line number may be affected by `#line` directives ([Line directives](#)).

The `CallerFilePath` attribute

The `System.Runtime.CompilerServices.CallerFilePathAttribute` is allowed on optional parameters when there is a standard implicit conversion ([Standard implicit conversions](#)) from `string` to the parameter's type.

If a function invocation from a location in source code omits an optional parameter with the `CallerFilePathAttribute`, then a string literal representing that location's file path is used as an argument to the invocation instead of the default parameter value.

The format of the file path is implementation-dependent.

Note that the file path may be affected by `#line` directives ([Line directives](#)).

The `CallerMemberName` attribute

The `System.Runtime.CompilerServices.CallerMemberNameAttribute` is allowed on optional parameters when there is a standard implicit conversion ([Standard implicit conversions](#)) from `string` to the parameter's type.

If a function invocation from a location within the body of a function member or within an attribute applied to the function member itself or its return type, parameters or type parameters in source code omits an optional parameter with the `CallerMemberNameAttribute`, then a string literal representing the name of that member is used as an argument to the invocation instead of the default parameter value.

For invocations that occur within generic methods, only the method name itself is used, without the type parameter list.

For invocations that occur within explicit interface member implementations, only the method name itself is used, without the preceding interface qualification.

For invocations that occur within property or event accessors, the member name used is that of the property or event itself.

For invocations that occur within indexer accessors, the member name used is that supplied by an `IndexerNameAttribute` (The [IndexerName attribute](#)) on the indexer member, if present, or the default name `Item` otherwise.

For invocations that occur within declarations of instance constructors, static constructors, destructors and operators the member name used is implementation-dependent.

Attributes for Interoperation

Note: This section is applicable only to the Microsoft .NET implementation of C#.

Interoperation with COM and Win32 components

The .NET run-time provides a large number of attributes that enable C# programs to interoperate with components written using COM and Win32 DLLs. For example, the `DllImport` attribute can be used on a `static extern` method to indicate that the implementation of the method is to be found in a Win32 DLL. These attributes are found in the `System.Runtime.InteropServices` namespace, and detailed documentation for these attributes is found in the .NET runtime documentation.

Interoperation with other .NET languages

The `IndexerName` attribute

Indexers are implemented in .NET using indexed properties, and have a name in the .NET metadata. If no `IndexerName` attribute is present for an indexer, then the name `Item` is used by default. The `IndexerName` attribute enables a developer to override this default and specify a different name.

```
namespace System.Runtime.CompilerServices.CSharp
{
    [AttributeUsage(AttributeTargets.Property)]
    public class IndexerNameAttribute: Attribute
    {
        public IndexerNameAttribute(string indexerName) {...}
        public string Value { get {...} }
    }
}
```

Unsafe code

12/28/2021 • 39 minutes to read • [Edit Online](#)

The core C# language, as defined in the preceding chapters, differs notably from C and C++ in its omission of pointers as a data type. Instead, C# provides references and the ability to create objects that are managed by a garbage collector. This design, coupled with other features, makes C# a much safer language than C or C++. In the core C# language it is simply not possible to have an uninitialized variable, a "dangling" pointer, or an expression that indexes an array beyond its bounds. Whole categories of bugs that routinely plague C and C++ programs are thus eliminated.

While practically every pointer type construct in C or C++ has a reference type counterpart in C#, nonetheless, there are situations where access to pointer types becomes a necessity. For example, interfacing with the underlying operating system, accessing a memory-mapped device, or implementing a time-critical algorithm may not be possible or practical without access to pointers. To address this need, C# provides the ability to write *unsafe code*.

In unsafe code it is possible to declare and operate on pointers, to perform conversions between pointers and integral types, to take the address of variables, and so forth. In a sense, writing unsafe code is much like writing C code within a C# program.

Unsafe code is in fact a "safe" feature from the perspective of both developers and users. Unsafe code must be clearly marked with the modifier `unsafe`, so developers can't possibly use unsafe features accidentally, and the execution engine works to ensure that unsafe code cannot be executed in an untrusted environment.

Unsafe contexts

The unsafe features of C# are available only in unsafe contexts. An unsafe context is introduced by including an `unsafe` modifier in the declaration of a type or member, or by employing an *unsafe_statement*.

- A declaration of a class, struct, interface, or delegate may include an `unsafe` modifier, in which case the entire textual extent of that type declaration (including the body of the class, struct, or interface) is considered an unsafe context.
- A declaration of a field, method, property, event, indexer, operator, instance constructor, destructor, or static constructor may include an `unsafe` modifier, in which case the entire textual extent of that member declaration is considered an unsafe context.
- An *unsafe_statement* enables the use of an unsafe context within a *block*. The entire textual extent of the associated *block* is considered an unsafe context.

The associated grammar productions are shown below.

```

class_modifier_unsafe
    : 'unsafe'
    ;

struct_modifier_unsafe
    : 'unsafe'
    ;

interface_modifier_unsafe
    : 'unsafe'
    ;

delegate_modifier_unsafe
    : 'unsafe'
    ;

field_modifier_unsafe
    : 'unsafe'
    ;

method_modifier_unsafe
    : 'unsafe'
    ;

property_modifier_unsafe
    : 'unsafe'
    ;

event_modifier_unsafe
    : 'unsafe'
    ;

indexer_modifier_unsafe
    : 'unsafe'
    ;

operator_modifier_unsafe
    : 'unsafe'
    ;

constructor_modifier_unsafe
    : 'unsafe'
    ;

destructor_declaration_unsafe
    : attributes? 'extern'? 'unsafe'? '~' identifier '(' ' ' )' destructor_body
    | attributes? 'unsafe'? 'extern'? '~' identifier '(' ' ' )' destructor_body
    ;

static_constructor_modifiers_unsafe
    : 'extern'? 'unsafe'? 'static'
    | 'unsafe'? 'extern'? 'static'
    | 'extern'? 'static' 'unsafe'?
    | 'unsafe'? 'static' 'extern'?
    | 'static' 'extern'? 'unsafe'?
    | 'static' 'unsafe'? 'extern'?
    ;

embedded_statement_unsafe
    : unsafe_statement
    | fixed_statement
    ;

unsafe_statement
    : 'unsafe' block
    ;

```


In the example

```
public unsafe struct Node
{
    public int Value;
    public Node* Left;
    public Node* Right;
}
```

the `unsafe` modifier specified in the struct declaration causes the entire textual extent of the struct declaration to become an unsafe context. Thus, it is possible to declare the `Left` and `Right` fields to be of a pointer type. The example above could also be written

```
public struct Node
{
    public int Value;
    public unsafe Node* Left;
    public unsafe Node* Right;
}
```

Here, the `unsafe` modifiers in the field declarations cause those declarations to be considered unsafe contexts.

Other than establishing an unsafe context, thus permitting the use of pointer types, the `unsafe` modifier has no effect on a type or a member. In the example

```
public class A
{
    public unsafe virtual void F() {
        char* p;
        ...
    }
}

public class B: A
{
    public override void F() {
        base.F();
        ...
    }
}
```

the `unsafe` modifier on the `F` method in `A` simply causes the textual extent of `F` to become an unsafe context in which the unsafe features of the language can be used. In the override of `F` in `B`, there is no need to re-specify the `unsafe` modifier -- unless, of course, the `F` method in `B` itself needs access to unsafe features.

The situation is slightly different when a pointer type is part of the method's signature

```
public unsafe class A
{
    public virtual void F(char* p) {...}
}

public class B: A
{
    public unsafe override void F(char* p) {...}
}
```

Here, because `F`'s signature includes a pointer type, it can only be written in an unsafe context. However, the unsafe context can be introduced by either making the entire class unsafe, as is the case in `A`, or by including an

`unsafe` modifier in the method declaration, as is the case in `B`.

Pointer types

In an unsafe context, a *type* ([Types](#)) may be a *pointer_type* as well as a *value_type* or a *reference_type*. However, a *pointer_type* may also be used in a `typeof` expression ([Anonymous object creation expressions](#)) outside of an unsafe context as such usage is not unsafe.

```
type_unsafe
    : pointer_type
    ;
```

A *pointer_type* is written as an *unmanaged_type* or the keyword `void`, followed by a `*` token:

```
pointer_type
    : unmanaged_type '*'
    | 'void' '*'
    ;

unmanaged_type
    : type
    ;
```

The type specified before the `*` in a pointer type is called the *referent type* of the pointer type. It represents the type of the variable to which a value of the pointer type points.

Unlike references (values of reference types), pointers are not tracked by the garbage collector -- the garbage collector has no knowledge of pointers and the data to which they point. For this reason a pointer is not permitted to point to a reference or to a struct that contains references, and the referent type of a pointer must be an *unmanaged_type*.

An *unmanaged_type* is any type that isn't a *reference_type* or constructed type, and doesn't contain *reference_type* or constructed type fields at any level of nesting. In other words, an *unmanaged_type* is one of the following:

- `sbyte`, `byte`, `short`, `ushort`, `int`, `uint`, `long`, `ulong`, `char`, `float`, `double`, `decimal`, Or `bool`.
- Any *enum_type*.
- Any *pointer_type*.
- Any user-defined *struct_type* that is not a constructed type and contains fields of *unmanaged_types* only.

The intuitive rule for mixing of pointers and references is that referents of references (objects) are permitted to contain pointers, but referents of pointers are not permitted to contain references.

Some examples of pointer types are given in the table below:

EXAMPLE	DESCRIPTION
<code>byte*</code>	Pointer to <code>byte</code>
<code>char*</code>	Pointer to <code>char</code>
<code>int**</code>	Pointer to pointer to <code>int</code>
<code>int*[]</code>	Single-dimensional array of pointers to <code>int</code>

EXAMPLE	DESCRIPTION
<code>void*</code>	Pointer to unknown type

For a given implementation, all pointer types must have the same size and representation.

Unlike C and C++, when multiple pointers are declared in the same declaration, in C# the `*` is written along with the underlying type only, not as a prefix punctuator on each pointer name. For example

```
int* pi, pj;    // NOT as int *pi, *pj;
```

The value of a pointer having type `T*` represents the address of a variable of type `T`. The pointer indirection operator `*` ([Pointer indirection](#)) may be used to access this variable. For example, given a variable `p` of type `int*`, the expression `*p` denotes the `int` variable found at the address contained in `p`.

Like an object reference, a pointer may be `null`. Applying the indirection operator to a `null` pointer results in implementation-defined behavior. A pointer with value `null` is represented by all-bits-zero.

The `void*` type represents a pointer to an unknown type. Because the referent type is unknown, the indirection operator cannot be applied to a pointer of type `void*`, nor can any arithmetic be performed on such a pointer. However, a pointer of type `void*` can be cast to any other pointer type (and vice versa).

Pointer types are a separate category of types. Unlike reference types and value types, pointer types do not inherit from `object` and no conversions exist between pointer types and `object`. In particular, boxing and unboxing ([Boxing and unboxing](#)) are not supported for pointers. However, conversions are permitted between different pointer types and between pointer types and the integral types. This is described in [Pointer conversions](#).

A *pointer_type* cannot be used as a type argument ([Constructed types](#)), and type inference ([Type inference](#)) fails on generic method calls that would have inferred a type argument to be a pointer type.

A *pointer_type* may be used as the type of a volatile field ([Volatile fields](#)).

Although pointers can be passed as `ref` or `out` parameters, doing so can cause undefined behavior, since the pointer may well be set to point to a local variable which no longer exists when the called method returns, or the fixed object to which it used to point, is no longer fixed. For example:

```

using System;

class Test
{
    static int value = 20;

    unsafe static void F(out int* pi1, ref int* pi2) {
        int i = 10;
        pi1 = &i;

        fixed (int* pj = &value) {
            // ...
            pi2 = pj;
        }
    }

    static void Main() {
        int i = 10;
        unsafe {
            int* px1;
            int* px2 = &i;

            F(out px1, ref px2);

            Console.WriteLine("*px1 = {0}, *px2 = {1}",
                              *px1, *px2);    // undefined behavior
        }
    }
}

```

A method can return a value of some type, and that type can be a pointer. For example, when given a pointer to a contiguous sequence of `int` s, that sequence's element count, and some other `int` value, the following method returns the address of that value in that sequence, if a match occurs; otherwise it returns `null` :

```

unsafe static int* Find(int* pi, int size, int value) {
    for (int i = 0; i < size; ++i) {
        if (*pi == value)
            return pi;
        ++pi;
    }
    return null;
}

```

In an unsafe context, several constructs are available for operating on pointers:

- The `*` operator may be used to perform pointer indirection ([Pointer indirection](#)).
- The `->` operator may be used to access a member of a struct through a pointer ([Pointer member access](#)).
- The `[]` operator may be used to index a pointer ([Pointer element access](#)).
- The `&` operator may be used to obtain the address of a variable ([The address-of operator](#)).
- The `++` and `--` operators may be used to increment and decrement pointers ([Pointer increment and decrement](#)).
- The `+` and `-` operators may be used to perform pointer arithmetic ([Pointer arithmetic](#)).
- The `==`, `!=`, `<`, `>`, `<=`, and `>=` operators may be used to compare pointers ([Pointer comparison](#)).
- The `stackalloc` operator may be used to allocate memory from the call stack ([Fixed size buffers](#)).
- The `fixed` statement may be used to temporarily fix a variable so its address can be obtained ([The fixed statement](#)).

Fixed and moveable variables

The address-of operator ([The address-of operator](#)) and the `fixed` statement ([The fixed statement](#)) divide variables into two categories: *Fixed variables* and *moveable variables*.

Fixed variables reside in storage locations that are unaffected by operation of the garbage collector. (Examples of fixed variables include local variables, value parameters, and variables created by dereferencing pointers.) On the other hand, moveable variables reside in storage locations that are subject to relocation or disposal by the garbage collector. (Examples of moveable variables include fields in objects and elements of arrays.)

The `&` operator ([The address-of operator](#)) permits the address of a fixed variable to be obtained without restrictions. However, because a moveable variable is subject to relocation or disposal by the garbage collector, the address of a moveable variable can only be obtained using a `fixed` statement ([The fixed statement](#)), and that address remains valid only for the duration of that `fixed` statement.

In precise terms, a fixed variable is one of the following:

- A variable resulting from a *simple_name* ([Simple names](#)) that refers to a local variable or a value parameter, unless the variable is captured by an anonymous function.
- A variable resulting from a *member_access* ([Member access](#)) of the form `v.I`, where `v` is a fixed variable of a *struct_type*.
- A variable resulting from a *pointer_indirection_expression* ([Pointer indirection](#)) of the form `*p`, a *pointer_member_access* ([Pointer member access](#)) of the form `p->I`, or a *pointer_element_access* ([Pointer element access](#)) of the form `p[E]`.

All other variables are classified as moveable variables.

Note that a static field is classified as a moveable variable. Also note that a `ref` or `out` parameter is classified as a moveable variable, even if the argument given for the parameter is a fixed variable. Finally, note that a variable produced by dereferencing a pointer is always classified as a fixed variable.

Pointer conversions

In an unsafe context, the set of available implicit conversions ([Implicit conversions](#)) is extended to include the following implicit pointer conversions:

- From any *pointer_type* to the type `void*`.
- From the `null` literal to any *pointer_type*.

Additionally, in an unsafe context, the set of available explicit conversions ([Explicit conversions](#)) is extended to include the following explicit pointer conversions:

- From any *pointer_type* to any other *pointer_type*.
- From `sbyte`, `byte`, `short`, `ushort`, `int`, `uint`, `long`, or `ulong` to any *pointer_type*.
- From any *pointer_type* to `sbyte`, `byte`, `short`, `ushort`, `int`, `uint`, `long`, or `ulong`.

Finally, in an unsafe context, the set of standard implicit conversions ([Standard implicit conversions](#)) includes the following pointer conversion:

- From any *pointer_type* to the type `void*`.

Conversions between two pointer types never change the actual pointer value. In other words, a conversion from one pointer type to another has no effect on the underlying address given by the pointer.

When one pointer type is converted to another, if the resulting pointer is not correctly aligned for the pointed-to type, the behavior is undefined if the result is dereferenced. In general, the concept "correctly aligned" is transitive: if a pointer to type `A` is correctly aligned for a pointer to type `B`, which, in turn, is correctly aligned for a pointer to type `C`, then a pointer to type `A` is correctly aligned for a pointer to type `C`.

Consider the following case in which a variable having one type is accessed via a pointer to a different type:

```
char c = 'A';
char* pc = &c;
void* pv = pc;
int* pi = (int*)pv;
int i = *pi;          // undefined
*pi = 123456;         // undefined
```

When a pointer type is converted to a pointer to byte, the result points to the lowest addressed byte of the variable. Successive increments of the result, up to the size of the variable, yield pointers to the remaining bytes of that variable. For example, the following method displays each of the eight bytes in a double as a hexadecimal value:

```
using System;

class Test
{
    unsafe static void Main() {
        double d = 123.456e23;
        unsafe {
            byte* pb = (byte*)&d;
            for (int i = 0; i < sizeof(double); ++i)
                Console.WriteLine("{0:X2} ", *pb++);
        }
    }
}
```

Of course, the output produced depends on endianness.

Mappings between pointers and integers are implementation-defined. However, on 32* and 64-bit CPU architectures with a linear address space, conversions of pointers to or from integral types typically behave exactly like conversions of `uint` or `ulong` values, respectively, to or from those integral types.

Pointer arrays

In an unsafe context, arrays of pointers can be constructed. Only some of the conversions that apply to other array types are allowed on pointer arrays:

- The implicit reference conversion ([Implicit reference conversions](#)) from any *array_type* to `System.Array` and the interfaces it implements also applies to pointer arrays. However, any attempt to access the array elements through `System.Array` or the interfaces it implements will result in an exception at run-time, as pointer types are not convertible to `object`.
- The implicit and explicit reference conversions ([Implicit reference conversions](#), [Explicit reference conversions](#)) from a single-dimensional array type `S[]` to `System.Collections.Generic.IList<T>` and its generic base interfaces never apply to pointer arrays, since pointer types cannot be used as type arguments, and there are no conversions from pointer types to non-pointer types.
- The explicit reference conversion ([Explicit reference conversions](#)) from `System.Array` and the interfaces it implements to any *array_type* applies to pointer arrays.
- The explicit reference conversions ([Explicit reference conversions](#)) from `System.Collections.Generic.IList<S>` and its base interfaces to a single-dimensional array type `T[]` never applies to pointer arrays, since pointer types cannot be used as type arguments, and there are no conversions from pointer types to non-pointer types.

These restrictions mean that the expansion for the `foreach` statement over arrays described in [The foreach statement](#) cannot be applied to pointer arrays. Instead, a `foreach` statement of the form

```
foreach (V v in x) embedded_statement
```

where the type of `x` is an array type of the form `T[,,...,]`, `N` is the number of dimensions minus 1 and `T` or `V` is a pointer type, is expanded using nested for-loops as follows:

```
{
    T[,,...,] a = x;
    for (int i0 = a.GetLowerBound(0); i0 <= a.GetUpperBound(0); i0++)
    for (int i1 = a.GetLowerBound(1); i1 <= a.GetUpperBound(1); i1++)
    ...
    for (int iN = a.GetLowerBound(N); iN <= a.GetUpperBound(N); iN++) {
        V v = (V)a.GetValue(i0,i1,...,iN);
        embedded_statement
    }
}
```

The variables `a`, `i0`, `i1`, ..., `iN` are not visible to or accessible to `x` or the *embedded_statement* or any other source code of the program. The variable `v` is read-only in the embedded statement. If there is not an explicit conversion ([Pointer conversions](#)) from `T` (the element type) to `V`, an error is produced and no further steps are taken. If `x` has the value `null`, a `System.NullReferenceException` is thrown at run-time.

Pointers in expressions

In an unsafe context, an expression may yield a result of a pointer type, but outside an unsafe context it is a compile-time error for an expression to be of a pointer type. In precise terms, outside an unsafe context a compile-time error occurs if any *simple_name* ([Simple names](#)), *member_access* ([Member access](#)), *invocation_expression* ([Invocation expressions](#)), or *element_access* ([Element access](#)) is of a pointer type.

In an unsafe context, the *primary_no_array_creation_expression* ([Primary expressions](#)) and *unary_expression* ([Unary operators](#)) productions permit the following additional constructs:

```
primary_no_array_creation_expression_unsafe
: pointer_member_access
| pointer_element_access
| sizeof_expression
;

unary_expression_unsafe
: pointer_indirection_expression
| addressof_expression
;
```

These constructs are described in the following sections. The precedence and associativity of the unsafe operators is implied by the grammar.

Pointer indirection

A *pointer_indirection_expression* consists of an asterisk (`*`) followed by a *unary_expression*.

```
pointer_indirection_expression
: '*' unary_expression
;
```

The unary `*` operator denotes pointer indirection and is used to obtain the variable to which a pointer points. The result of evaluating `*P`, where `P` is an expression of a pointer type `T*`, is a variable of type `T`. It is a compile-time error to apply the unary `*` operator to an expression of type `void*` or to an expression that isn't

of a pointer type.

The effect of applying the unary `*` operator to a `null` pointer is implementation-defined. In particular, there is no guarantee that this operation throws a `System.NullReferenceException`.

If an invalid value has been assigned to the pointer, the behavior of the unary `*` operator is undefined. Among the invalid values for dereferencing a pointer by the unary `*` operator are an address inappropriately aligned for the type pointed to (see example in [Pointer conversions](#)), and the address of a variable after the end of its lifetime.

For purposes of definite assignment analysis, a variable produced by evaluating an expression of the form `*P` is considered initially assigned ([Initially assigned variables](#)).

Pointer member access

A *pointer_member_access* consists of a *primary_expression*, followed by a `->` token, followed by an *identifier* and an optional *type_argument_list*.

```
pointer_member_access
    : primary_expression '->' identifier
    ;
```

In a pointer member access of the form `P->I`, `P` must be an expression of a pointer type other than `void*`, and `I` must denote an accessible member of the type to which `P` points.

A pointer member access of the form `P->I` is evaluated exactly as `(*P).I`. For a description of the pointer indirection operator (`*`), see [Pointer indirection](#). For a description of the member access operator (`.`), see [Member access](#).

In the example

```
using System;

struct Point
{
    public int x;
    public int y;

    public override string ToString() {
        return "(" + x + "," + y + ")";
    }
}

class Test
{
    static void Main() {
        Point point;
        unsafe {
            Point* p = &point;
            p->x = 10;
            p->y = 20;
            Console.WriteLine(p->ToString());
        }
    }
}
```

the `->` operator is used to access fields and invoke a method of a struct through a pointer. Because the operation `P->I` is precisely equivalent to `(*P).I`, the `Main` method could equally well have been written:


```

class Test
{
    static void Main() {
        Point point;
        unsafe {
            Point* p = &point;
            (*p).x = 10;
            (*p).y = 20;
            Console.WriteLine((*p).ToString());
        }
    }
}

```

Pointer element access

A *pointer_element_access* consists of a *primary_no_array_creation_expression* followed by an expression enclosed in "[" and "]".

```

pointer_element_access
: primary_no_array_creation_expression '[' expression ']'
;

```

In a pointer element access of the form `P[E]`, `P` must be an expression of a pointer type other than `void*`, and `E` must be an expression that can be implicitly converted to `int`, `uint`, `long`, or `ulong`.

A pointer element access of the form `P[E]` is evaluated exactly as `*(P + E)`. For a description of the pointer indirection operator (`*`), see [Pointer indirection](#). For a description of the pointer addition operator (`+`), see [Pointer arithmetic](#).

In the example

```

class Test
{
    static void Main() {
        unsafe {
            char* p = stackalloc char[256];
            for (int i = 0; i < 256; i++) p[i] = (char)i;
        }
    }
}

```

a pointer element access is used to initialize the character buffer in a `for` loop. Because the operation `P[E]` is precisely equivalent to `*(P + E)`, the example could equally well have been written:

```

class Test
{
    static void Main() {
        unsafe {
            char* p = stackalloc char[256];
            for (int i = 0; i < 256; i++) *(p + i) = (char)i;
        }
    }
}

```

The pointer element access operator does not check for out-of-bounds errors and the behavior when accessing an out-of-bounds element is undefined. This is the same as C and C++.

The address-of operator

An *addressof_expression* consists of an ampersand (`&`) followed by a *unary_expression*.

```
addressof_expression
: '&' unary_expression
;
```

Given an expression `E` which is of a type `T` and is classified as a fixed variable ([Fixed and moveable variables](#)), the construct `&E` computes the address of the variable given by `E`. The type of the result is `T*` and is classified as a value. A compile-time error occurs if `E` is not classified as a variable, if `E` is classified as a read-only local variable, or if `E` denotes a moveable variable. In the last case, a fixed statement ([The fixed statement](#)) can be used to temporarily "fix" the variable before obtaining its address. As stated in [Member access](#), outside an instance constructor or static constructor for a struct or class that defines a `readonly` field, that field is considered a value, not a variable. As such, its address cannot be taken. Similarly, the address of a constant cannot be taken.

The `&` operator does not require its argument to be definitely assigned, but following an `&` operation, the variable to which the operator is applied is considered definitely assigned in the execution path in which the operation occurs. It is the responsibility of the programmer to ensure that correct initialization of the variable actually does take place in this situation.

In the example

```
using System;

class Test
{
    static void Main() {
        int i;
        unsafe {
            int* p = &i;
            *p = 123;
        }
        Console.WriteLine(i);
    }
}
```

`i` is considered definitely assigned following the `&i` operation used to initialize `p`. The assignment to `*p` in effect initializes `i`, but the inclusion of this initialization is the responsibility of the programmer, and no compile-time error would occur if the assignment was removed.

The rules of definite assignment for the `&` operator exist such that redundant initialization of local variables can be avoided. For example, many external APIs take a pointer to a structure which is filled in by the API. Calls to such APIs typically pass the address of a local struct variable, and without the rule, redundant initialization of the struct variable would be required.

Pointer increment and decrement

In an unsafe context, the `++` and `--` operators ([Postfix increment and decrement operators](#) and [Prefix increment and decrement operators](#)) can be applied to pointer variables of all types except `void*`. Thus, for every pointer type `T*`, the following operators are implicitly defined:

```
T* operator ++(T* x);
T* operator --(T* x);
```

The operators produce the same results as `x + 1` and `x - 1`, respectively ([Pointer arithmetic](#)). In other words, for a pointer variable of type `T*`, the `++` operator adds `sizeof(T)` to the address contained in the variable, and

the `--` operator subtracts `sizeof(T)` from the address contained in the variable.

If a pointer increment or decrement operation overflows the domain of the pointer type, the result is implementation-defined, but no exceptions are produced.

Pointer arithmetic

In an unsafe context, the `+` and `-` operators ([Addition operator](#) and [Subtraction operator](#)) can be applied to values of all pointer types except `void*`. Thus, for every pointer type `T*`, the following operators are implicitly defined:

```
T* operator +(T* x, int y);
T* operator +(T* x, uint y);
T* operator +(T* x, long y);
T* operator +(T* x, ulong y);

T* operator +(int x, T* y);
T* operator +(uint x, T* y);
T* operator +(long x, T* y);
T* operator +(ulong x, T* y);

T* operator -(T* x, int y);
T* operator -(T* x, uint y);
T* operator -(T* x, long y);
T* operator -(T* x, ulong y);

long operator -(T* x, T* y);
```

Given an expression `P` of a pointer type `T*` and an expression `N` of type `int`, `uint`, `long`, or `ulong`, the expressions `P + N` and `N + P` compute the pointer value of type `T*` that results from adding `N * sizeof(T)` to the address given by `P`. Likewise, the expression `P - N` computes the pointer value of type `T*` that results from subtracting `N * sizeof(T)` from the address given by `P`.

Given two expressions, `P` and `Q`, of a pointer type `T*`, the expression `P - Q` computes the difference between the addresses given by `P` and `Q` and then divides that difference by `sizeof(T)`. The type of the result is always `long`. In effect, `P - Q` is computed as `((long)(P) - (long)(Q)) / sizeof(T)`.

For example:

```
using System;

class Test
{
    static void Main() {
        unsafe {
            int* values = stackalloc int[20];
            int* p = &values[1];
            int* q = &values[15];
            Console.WriteLine("p - q = {0}", p - q);
            Console.WriteLine("q - p = {0}", q - p);
        }
    }
}
```

which produces the output:

```
p - q = -14
q - p = 14
```

If a pointer arithmetic operation overflows the domain of the pointer type, the result is truncated in an

implementation-defined fashion, but no exceptions are produced.

Pointer comparison

In an unsafe context, the `==`, `!=`, `<`, `>`, `<=`, and `>=` operators ([Relational and type-testing operators](#)) can be applied to values of all pointer types. The pointer comparison operators are:

```
bool operator ==(void* x, void* y);
bool operator !=(void* x, void* y);
bool operator <(void* x, void* y);
bool operator >(void* x, void* y);
bool operator <=(void* x, void* y);
bool operator >=(void* x, void* y);
```

Because an implicit conversion exists from any pointer type to the `void*` type, operands of any pointer type can be compared using these operators. The comparison operators compare the addresses given by the two operands as if they were unsigned integers.

The sizeof operator

The `sizeof` operator returns the number of bytes occupied by a variable of a given type. The type specified as an operand to `sizeof` must be an *unmanaged_type* ([Pointer types](#)).

```
sizeof_expression
: 'sizeof' '(' unmanaged_type ')'
;
```

The result of the `sizeof` operator is a value of type `int`. For certain predefined types, the `sizeof` operator yields a constant value as shown in the table below.

EXPRESSION	RESULT
<code>sizeof(sbyte)</code>	1
<code>sizeof(byte)</code>	1
<code>sizeof(short)</code>	2
<code>sizeof(ushort)</code>	2
<code>sizeof(int)</code>	4
<code>sizeof(uint)</code>	4
<code>sizeof(long)</code>	8
<code>sizeof(ulong)</code>	8
<code>sizeof(char)</code>	2
<code>sizeof(float)</code>	4
<code>sizeof(double)</code>	8

EXPRESSION	RESULT
<code>sizeof(bool)</code>	1

For all other types, the result of the `sizeof` operator is implementation-defined and is classified as a value, not a constant.

The order in which members are packed into a struct is unspecified.

For alignment purposes, there may be unnamed padding at the beginning of a struct, within a struct, and at the end of the struct. The contents of the bits used as padding are indeterminate.

When applied to an operand that has struct type, the result is the total number of bytes in a variable of that type, including any padding.

The fixed statement

In an unsafe context, the *embedded_statement* ([Statements](#)) production permits an additional construct, the `fixed` statement, which is used to "fix" a moveable variable such that its address remains constant for the duration of the statement.

```

fixed_statement
: 'fixed' '(' pointer_type fixed_pointer_declarators ')' embedded_statement
;

fixed_pointer_declarators
: fixed_pointer_declarator (',' fixed_pointer_declarator)*
;

fixed_pointer_declarator
: identifier '=' fixed_pointer_initializer
;

fixed_pointer_initializer
: '&' variable_reference
| expression
;

```

Each *fixed_pointer_declarator* declares a local variable of the given *pointer_type* and initializes that local variable with the address computed by the corresponding *fixed_pointer_initializer*. A local variable declared in a `fixed` statement is accessible in any *fixed_pointer_initializers* occurring to the right of that variable's declaration, and in the *embedded_statement* of the `fixed` statement. A local variable declared by a `fixed` statement is considered read-only. A compile-time error occurs if the embedded statement attempts to modify this local variable (via assignment or the `++` and `--` operators) or pass it as a `ref` or `out` parameter.

A *fixed_pointer_initializer* can be one of the following:

- The token `"&"` followed by a *variable_reference* ([Precise rules for determining definite assignment](#)) to a moveable variable ([Fixed and moveable variables](#)) of an unmanaged type `T`, provided the type `T*` is implicitly convertible to the pointer type given in the `fixed` statement. In this case, the initializer computes the address of the given variable, and the variable is guaranteed to remain at a fixed address for the duration of the `fixed` statement.
- An expression of an *array_type* with elements of an unmanaged type `T`, provided the type `T*` is implicitly convertible to the pointer type given in the `fixed` statement. In this case, the initializer computes the address of the first element in the array, and the entire array is guaranteed to remain at a fixed address for the duration of the `fixed` statement. If the array expression is null or if the array has zero elements, the initializer computes an address equal to zero.

- An expression of type `string`, provided the type `char*` is implicitly convertible to the pointer type given in the `fixed` statement. In this case, the initializer computes the address of the first character in the string, and the entire string is guaranteed to remain at a fixed address for the duration of the `fixed` statement. The behavior of the `fixed` statement is implementation-defined if the string expression is null.
- A *simple_name* or *member_access* that references a fixed size buffer member of a moveable variable, provided the type of the fixed size buffer member is implicitly convertible to the pointer type given in the `fixed` statement. In this case, the initializer computes a pointer to the first element of the fixed size buffer ([Fixed size buffers in expressions](#)), and the fixed size buffer is guaranteed to remain at a fixed address for the duration of the `fixed` statement.

For each address computed by a *fixed_pointer_initializer* the `fixed` statement ensures that the variable referenced by the address is not subject to relocation or disposal by the garbage collector for the duration of the `fixed` statement. For example, if the address computed by a *fixed_pointer_initializer* references a field of an object or an element of an array instance, the `fixed` statement guarantees that the containing object instance is not relocated or disposed of during the lifetime of the statement.

It is the programmer's responsibility to ensure that pointers created by `fixed` statements do not survive beyond execution of those statements. For example, when pointers created by `fixed` statements are passed to external APIs, it is the programmer's responsibility to ensure that the APIs retain no memory of these pointers.

Fixed objects may cause fragmentation of the heap (because they can't be moved). For that reason, objects should be fixed only when absolutely necessary and then only for the shortest amount of time possible.

The example

```
class Test
{
    static int x;
    int y;

    unsafe static void F(int* p) {
        *p = 1;
    }

    static void Main() {
        Test t = new Test();
        int[] a = new int[10];
        unsafe {
            fixed (int* p = &x) F(p);
            fixed (int* p = &t.y) F(p);
            fixed (int* p = &a[0]) F(p);
            fixed (int* p = a) F(p);
        }
    }
}
```

demonstrates several uses of the `fixed` statement. The first statement fixes and obtains the address of a static field, the second statement fixes and obtains the address of an instance field, and the third statement fixes and obtains the address of an array element. In each case it would have been an error to use the regular `&` operator since the variables are all classified as moveable variables.

The fourth `fixed` statement in the example above produces a similar result to the third.

This example of the `fixed` statement uses `string`:

```

class Test
{
    static string name = "xx";

    unsafe static void F(char* p) {
        for (int i = 0; p[i] != '\0'; ++i)
            Console.WriteLine(p[i]);
    }

    static void Main() {
        unsafe {
            fixed (char* p = name) F(p);
            fixed (char* p = "xx") F(p);
        }
    }
}

```

In an unsafe context array elements of single-dimensional arrays are stored in increasing index order, starting with index `0` and ending with index `Length - 1`. For multi-dimensional arrays, array elements are stored such that the indices of the rightmost dimension are increased first, then the next left dimension, and so on to the left. Within a `fixed` statement that obtains a pointer `p` to an array instance `a`, the pointer values ranging from `p` to `p + a.Length - 1` represent addresses of the elements in the array. Likewise, the variables ranging from `p[0]` to `p[a.Length - 1]` represent the actual array elements. Given the way in which arrays are stored, we can treat an array of any dimension as though it were linear.

For example:

```

using System;

class Test
{
    static void Main() {
        int[, ,] a = new int[2,3,4];
        unsafe {
            fixed (int* p = a) {
                for (int i = 0; i < a.Length; ++i)    // treat as linear
                    p[i] = i;
            }
        }

        for (int i = 0; i < 2; ++i)
            for (int j = 0; j < 3; ++j) {
                for (int k = 0; k < 4; ++k)
                    Console.Write("{0},{1},{2}] = {3,2} ", i, j, k, a[i,j,k]);
                Console.WriteLine();
            }
    }
}

```

which produces the output:

```

[0,0,0] = 0 [0,0,1] = 1 [0,0,2] = 2 [0,0,3] = 3
[0,1,0] = 4 [0,1,1] = 5 [0,1,2] = 6 [0,1,3] = 7
[0,2,0] = 8 [0,2,1] = 9 [0,2,2] = 10 [0,2,3] = 11
[1,0,0] = 12 [1,0,1] = 13 [1,0,2] = 14 [1,0,3] = 15
[1,1,0] = 16 [1,1,1] = 17 [1,1,2] = 18 [1,1,3] = 19
[1,2,0] = 20 [1,2,1] = 21 [1,2,2] = 22 [1,2,3] = 23

```

In the example

```

class Test
{
    unsafe static void Fill(int* p, int count, int value) {
        for (; count != 0; count--) *p++ = value;
    }

    static void Main() {
        int[] a = new int[100];
        unsafe {
            fixed (int* p = a) Fill(p, 100, -1);
        }
    }
}

```

a `fixed` statement is used to fix an array so its address can be passed to a method that takes a pointer.

In the example:

```

unsafe struct Font
{
    public int size;
    public fixed char name[32];
}

class Test
{
    unsafe static void PutString(string s, char* buffer, int bufSize) {
        int len = s.Length;
        if (len > bufSize) len = bufSize;
        for (int i = 0; i < len; i++) buffer[i] = s[i];
        for (int i = len; i < bufSize; i++) buffer[i] = (char)0;
    }

    Font f;

    unsafe static void Main()
    {
        Test test = new Test();
        test.f.size = 10;
        fixed (char* p = test.f.name) {
            PutString("Times New Roman", p, 32);
        }
    }
}

```

a `fixed` statement is used to fix a fixed size buffer of a struct so its address can be used as a pointer.

A `char*` value produced by fixing a string instance always points to a null-terminated string. Within a `fixed` statement that obtains a pointer `p` to a string instance `s`, the pointer values ranging from `p` to `p + s.Length - 1` represent addresses of the characters in the string, and the pointer value `p + s.Length` always points to a null character (the character with value `'\0'`).

Modifying objects of managed type through fixed pointers can result in undefined behavior. For example, because strings are immutable, it is the programmer's responsibility to ensure that the characters referenced by a pointer to a fixed string are not modified.

The automatic null-termination of strings is particularly convenient when calling external APIs that expect "C-style" strings. Note, however, that a string instance is permitted to contain null characters. If such null characters are present, the string will appear truncated when treated as a null-terminated `char*`.

Fixed size buffers

Fixed size buffers are used to declare "C style" in-line arrays as members of structs, and are primarily useful for interfacing with unmanaged APIs.

Fixed size buffer declarations

A *fixed size buffer* is a member that represents storage for a fixed length buffer of variables of a given type. A fixed size buffer declaration introduces one or more fixed size buffers of a given element type. Fixed size buffers are only permitted in struct declarations and can only occur in unsafe contexts ([Unsafe contexts](#)).

```
struct_member_declaration_unsafe
: fixed_size_buffer_declaration
;

fixed_size_buffer_declaration
: attributes? fixed_size_buffer_modifier* 'fixed' buffer_element_type fixed_size_buffer_declarator+ ';'
;

fixed_size_buffer_modifier
: 'new'
| 'public'
| 'protected'
| 'internal'
| 'private'
| 'unsafe'
;

buffer_element_type
: type
;

fixed_size_buffer_declarator
: identifier '[' constant_expression ']'
;
```

A fixed size buffer declaration may include a set of attributes ([Attributes](#)), a `new` modifier ([Modifiers](#)), a valid combination of the four access modifiers ([Type parameters and constraints](#)) and an `unsafe` modifier ([Unsafe contexts](#)). The attributes and modifiers apply to all of the members declared by the fixed size buffer declaration. It is an error for the same modifier to appear multiple times in a fixed size buffer declaration.

A fixed size buffer declaration is not permitted to include the `static` modifier.

The buffer element type of a fixed size buffer declaration specifies the element type of the buffer(s) introduced by the declaration. The buffer element type must be one of the predefined types `sbyte`, `byte`, `short`, `ushort`, `int`, `uint`, `long`, `ulong`, `char`, `float`, `double`, or `bool`.

The buffer element type is followed by a list of fixed size buffer declarators, each of which introduces a new member. A fixed size buffer declarator consists of an identifier that names the member, followed by a constant expression enclosed in `[` and `]` tokens. The constant expression denotes the number of elements in the member introduced by that fixed size buffer declarator. The type of the constant expression must be implicitly convertible to type `int`, and the value must be a non-zero positive integer.

The elements of a fixed size buffer are guaranteed to be laid out sequentially in memory.

A fixed size buffer declaration that declares multiple fixed size buffers is equivalent to multiple declarations of a single fixed size buffer declaration with the same attributes, and element types. For example

```
unsafe struct A
{
    public fixed int x[5], y[10], z[100];
}
```

is equivalent to

```
unsafe struct A
{
    public fixed int x[5];
    public fixed int y[10];
    public fixed int z[100];
}
```

Fixed size buffers in expressions

Member lookup ([Operators](#)) of a fixed size buffer member proceeds exactly like member lookup of a field.

A fixed size buffer can be referenced in an expression using a *simple_name* ([Type inference](#)) or a *member_access* ([Compile-time checking of dynamic overload resolution](#)).

When a fixed size buffer member is referenced as a simple name, the effect is the same as a member access of the form `this.I`, where `I` is the fixed size buffer member.

In a member access of the form `E.I`, if `E` is of a struct type and a member lookup of `I` in that struct type identifies a fixed size member, then `E.I` is evaluated and classified as follows:

- If the expression `E.I` does not occur in an unsafe context, a compile-time error occurs.
- If `E` is classified as a value, a compile-time error occurs.
- Otherwise, if `E` is a moveable variable ([Fixed and moveable variables](#)) and the expression `E.I` is not a *fixed_pointer_initializer* ([The fixed statement](#)), a compile-time error occurs.
- Otherwise, `E` references a fixed variable and the result of the expression is a pointer to the first element of the fixed size buffer member `I` in `E`. The result is of type `S*`, where `S` is the element type of `I`, and is classified as a value.

The subsequent elements of the fixed size buffer can be accessed using pointer operations from the first element. Unlike access to arrays, access to the elements of a fixed size buffer is an unsafe operation and is not range checked.

The following example declares and uses a struct with a fixed size buffer member.

```
unsafe struct Font
{
    public int size;
    public fixed char name[32];
}

class Test
{
    unsafe static void PutString(string s, char* buffer, int bufSize) {
        int len = s.Length;
        if (len > bufSize) len = bufSize;
        for (int i = 0; i < len; i++) buffer[i] = s[i];
        for (int i = len; i < bufSize; i++) buffer[i] = (char)0;
    }

    unsafe static void Main()
    {
        Font f;
        f.size = 10;
        PutString("Times New Roman", f.name, 32);
    }
}
```

Definite assignment checking

Fixed size buffers are not subject to definite assignment checking ([Definite assignment](#)), and fixed size buffer members are ignored for purposes of definite assignment checking of struct type variables.

When the outermost containing struct variable of a fixed size buffer member is a static variable, an instance variable of a class instance, or an array element, the elements of the fixed size buffer are automatically initialized to their default values ([Default values](#)). In all other cases, the initial content of a fixed size buffer is undefined.

Stack allocation

In an unsafe context, a local variable declaration ([Local variable declarations](#)) may include a stack allocation initializer which allocates memory from the call stack.

```
local_variable_initializer_unsafe
: stackalloc_initializer
;

stackalloc_initializer
: 'stackalloc' unmanaged_type '[' expression ']'
;
```

The *unmanaged_type* indicates the type of the items that will be stored in the newly allocated location, and the *expression* indicates the number of these items. Taken together, these specify the required allocation size. Since the size of a stack allocation cannot be negative, it is a compile-time error to specify the number of items as a *constant_expression* that evaluates to a negative value.

A stack allocation initializer of the form `stackalloc T[E]` requires `T` to be an unmanaged type ([Pointer types](#)) and `E` to be an expression of type `int`. The construct allocates `E * sizeof(T)` bytes from the call stack and returns a pointer, of type `T*`, to the newly allocated block. If `E` is a negative value, then the behavior is undefined. If `E` is zero, then no allocation is made, and the pointer returned is implementation-defined. If there is not enough memory available to allocate a block of the given size, a `System.StackOverflowException` is thrown.

The content of the newly allocated memory is undefined.

Stack allocation initializers are not permitted in `catch` or `finally` blocks ([The try statement](#)).

There is no way to explicitly free memory allocated using `stackalloc`. All stack allocated memory blocks created during the execution of a function member are automatically discarded when that function member returns. This corresponds to the `alloca` function, an extension commonly found in C and C++ implementations.

In the example

```

using System;

class Test
{
    static string IntToString(int value) {
        int n = value >= 0? value: -value;
        unsafe {
            char* buffer = stackalloc char[16];
            char* p = buffer + 16;
            do {
                *--p = (char)(n % 10 + '0');
                n /= 10;
            } while (n != 0);
            if (value < 0) *--p = '-';
            return new string(p, 0, (int)(buffer + 16 - p));
        }
    }

    static void Main() {
        Console.WriteLine(IntToString(12345));
        Console.WriteLine(IntToString(-999));
    }
}

```

a `stackalloc` initializer is used in the `IntToString` method to allocate a buffer of 16 characters on the stack. The buffer is automatically discarded when the method returns.

Dynamic memory allocation

Except for the `stackalloc` operator, C# provides no predefined constructs for managing non-garbage collected memory. Such services are typically provided by supporting class libraries or imported directly from the underlying operating system. For example, the `Memory` class below illustrates how the heap functions of an underlying operating system might be accessed from C#:

```

using System;
using System.Runtime.InteropServices;

public static unsafe class Memory
{
    // Handle for the process heap. This handle is used in all calls to the
    // HeapXXX APIs in the methods below.
    private static readonly IntPtr s_heap = GetProcessHeap();

    // Allocates a memory block of the given size. The allocated memory is
    // automatically initialized to zero.
    public static void* Alloc(int size)
    {
        void* result = HeapAlloc(s_heap, HEAP_ZERO_MEMORY, (UIntPtr)size);
        if (result == null) throw new OutOfMemoryException();
        return result;
    }

    // Copies count bytes from src to dst. The source and destination
    // blocks are permitted to overlap.
    public static void Copy(void* src, void* dst, int count)
    {
        byte* ps = (byte*)src;
        byte* pd = (byte*)dst;
        if (ps > pd)
        {
            for (; count != 0; count--) *pd++ = *ps++;
        }
        else if (ps < pd)
        {

```

```

        for (ps += count, pd += count; count != 0; count--) *--pd = *--ps;
    }
}

// Frees a memory block.
public static void Free(void* block)
{
    if (!HeapFree(s_heap, 0, block)) throw new InvalidOperationException();
}

// Re-allocates a memory block. If the reallocation request is for a
// larger size, the additional region of memory is automatically
// initialized to zero.
public static void* ReAlloc(void* block, int size)
{
    void* result = HeapReAlloc(s_heap, HEAP_ZERO_MEMORY, block, (UIntPtr)size);
    if (result == null) throw new OutOfMemoryException();
    return result;
}

// Returns the size of a memory block.
public static int SizeOf(void* block)
{
    int result = (int)HeapSize(s_heap, 0, block);
    if (result == -1) throw new InvalidOperationException();
    return result;
}

// Heap API flags
private const int HEAP_ZERO_MEMORY = 0x00000008;

// Heap API functions
[DllImport("kernel32")]
private static extern IntPtr GetProcessHeap();

[DllImport("kernel32")]
private static extern void* HeapAlloc(IntPtr hHeap, int flags, UIntPtr size);

[DllImport("kernel32")]
private static extern bool HeapFree(IntPtr hHeap, int flags, void* block);

[DllImport("kernel32")]
private static extern void* HeapReAlloc(IntPtr hHeap, int flags, void* block, UIntPtr size);

[DllImport("kernel32")]
private static extern UIntPtr HeapSize(IntPtr hHeap, int flags, void* block);
}

```

An example that uses the `Memory` class is given below:

```
class Test
{
    static unsafe void Main()
    {
        byte* buffer = null;
        try
        {
            const int Size = 256;
            buffer = (byte*)Memory.Alloc(Size);
            for (int i = 0; i < Size; i++) buffer[i] = (byte)i;
            byte[] array = new byte[Size];
            fixed (byte* p = array) Memory.Copy(buffer, p, Size);
            for (int i = 0; i < Size; i++) Console.WriteLine(array[i]);
        }
        finally
        {
            if (buffer != null) Memory.Free(buffer);
        }
    }
}
```

The example allocates 256 bytes of memory through `Memory.Alloc` and initializes the memory block with values increasing from 0 to 255. It then allocates a 256 element byte array and uses `Memory.Copy` to copy the contents of the memory block into the byte array. Finally, the memory block is freed using `Memory.Free` and the contents of the byte array are output on the console.

Documentation comments

12/28/2021 • 22 minutes to read • [Edit Online](#)

C# provides a mechanism for programmers to document their code using a special comment syntax that contains XML text. In source code files, comments having a certain form can be used to direct a tool to produce XML from those comments and the source code elements, which they precede. Comments using such syntax are called *documentation comments*. They must immediately precede a user-defined type (such as a class, delegate, or interface) or a member (such as a field, event, property, or method). The XML generation tool is called the *documentation generator*. (This generator could be, but need not be, the C# compiler itself.) The output produced by the documentation generator is called the *documentation file*. A documentation file is used as input to a *documentation viewer*, a tool intended to produce some sort of visual display of type information and its associated documentation.

This specification suggests a set of tags to be used in documentation comments, but use of these tags is not required, and other tags may be used if desired, as long the rules of well-formed XML are followed.

Introduction

Comments having a special form can be used to direct a tool to produce XML from those comments and the source code elements, which they precede. Such comments are single-line comments that start with three slashes (`///`), or delimited comments that start with a slash and two stars (`/**`). They must immediately precede a user-defined type (such as a class, delegate, or interface) or a member (such as a field, event, property, or method) that they annotate. Attribute sections ([Attribute specification](#)) are considered part of declarations, so documentation comments must precede attributes applied to a type or member.

Syntax:

```
single_line_doc_comment
    : '///' input_character*
    ;

delimited_doc_comment
    : '/**' delimited_comment_section* asterisk+ '/'
    ;
```

In a *single_line_doc_comment*, if there is a *whitespace* character following the `///` characters on each of the *single_line_doc_comments* adjacent to the current *single_line_doc_comment*, then that *whitespace* character is not included in the XML output.

In a delimited-doc-comment, if the first non-whitespace character on the second line is an asterisk and the same pattern of optional whitespace characters and an asterisk character is repeated at the beginning of each of the line within the delimited-doc-comment, then the characters of the repeated pattern are not included in the XML output. The pattern may include whitespace characters after, as well as before, the asterisk character.

Example:

```

/// <summary>Class <c>Point</c> models a point in a two-dimensional
/// plane.</summary>
///
public class Point
{
    /// <summary>method <c>draw</c> renders the point.</summary>
    void draw() {...}
}

```

The text within documentation comments must be well formed according to the rules of XML (<https://www.w3.org/TR/REC-xml>). If the XML is ill formed, a warning is generated and the documentation file will contain a comment saying that an error was encountered.

Although developers are free to create their own set of tags, a recommended set is defined in [Recommended tags](#). Some of the recommended tags have special meanings:

- The `<param>` tag is used to describe parameters. If such a tag is used, the documentation generator must verify that the specified parameter exists and that all parameters are described in documentation comments. If such verification fails, the documentation generator issues a warning.
- The `cref` attribute can be attached to any tag to provide a reference to a code element. The documentation generator must verify that this code element exists. If the verification fails, the documentation generator issues a warning. When looking for a name described in a `cref` attribute, the documentation generator must respect namespace visibility according to `using` statements appearing within the source code. For code elements that are generic, the normal generic syntax (that is, "`List<T>`") cannot be used because it produces invalid XML. Braces can be used instead of brackets (that is, "`List{T}`"), or the XML escape syntax can be used (that is, "`List<T>`").
- The `<summary>` tag is intended to be used by a documentation viewer to display additional information about a type or member.
- The `<include>` tag includes information from an external XML file.

Note carefully that the documentation file does not provide full information about the type and members (for example, it does not contain any type information). To get such information about a type or member, the documentation file must be used in conjunction with reflection on the actual type or member.

Recommended tags

The documentation generator must accept and process any tag that is valid according to the rules of XML. The following tags provide commonly used functionality in user documentation. (Of course, other tags are possible.)

TAG	SECTION	PURPOSE
<code><c></code>	<code><c></code>	Set text in a code-like font
<code><code></code>	<code><code></code>	Set one or more lines of source code or program output
<code><example></code>	<code><example></code>	Indicate an example
<code><exception></code>	<code><exception></code>	Identifies the exceptions a method can throw
<code><include></code>	<code><include></code>	Includes XML from an external file

TAG	SECTION	PURPOSE
<code><list></code>	<code><list></code>	Create a list or table
<code><para></code>	<code><para></code>	Permit structure to be added to text
<code><param></code>	<code><param></code>	Describe a parameter for a method or constructor
<code><paramref></code>	<code><paramref></code>	Identify that a word is a parameter name
<code><permission></code>	<code><permission></code>	Document the security accessibility of a member
<code><remarks></code>	<code><remarks></code>	Describe additional information about a type
<code><returns></code>	<code><returns></code>	Describe the return value of a method
<code><see></code>	<code><see></code>	Specify a link
<code><seealso></code>	<code><seealso></code>	Generate a See Also entry
<code><summary></code>	<code><summary></code>	Describe a type or a member of a type
<code><value></code>	<code><value></code>	Describe a property
<code><typeparam></code>		Describe a generic type parameter
<code><typeparamref></code>		Identify that a word is a type parameter name

`<c>`

This tag provides a mechanism to indicate that a fragment of text within a description should be set in a special font such as that used for a block of code. For lines of actual code, use `<code>` (`<code>`).

Syntax:

```
<c>text</c>
```

Example:

```
/// <summary>Class <c>Point</c> models a point in a two-dimensional
/// plane.</summary>

public class Point
{
    // ...
}
```

`<code>`

This tag is used to set one or more lines of source code or program output in some special font. For small code

fragments in narrative, use `<c>` (`<c>`).

Syntax:

```
<code>source code or program output</code>
```

Example:

```
/// <summary>This method changes the point's location by  
///     the given x- and y-offsets.  
/// <example>For example:  
/// <code>  
///     Point p = new Point(3,5);  
///     p.Translate(-1,3);  
/// </code>  
/// results in <c>p</c>'s having the value (2,8).  
/// </example>  
/// </summary>  
  
public void Translate(int xor, int yor) {  
    X += xor;  
    Y += yor;  
}
```

`<example>`

This tag allows example code within a comment, to specify how a method or other library member may be used. Ordinarily, this would also involve use of the tag `<code>` (`<code>`) as well.

Syntax:

```
<example>description</example>
```

Example:

See `<code>` (`<code>`) for an example.

`<exception>`

This tag provides a way to document the exceptions a method can throw.

Syntax:

```
<exception cref="member">description</exception>
```

where

- `member` is the name of a member. The documentation generator checks that the given member exists and translates `member` to the canonical element name in the documentation file.
- `description` is a description of the circumstances in which the exception is thrown.

Example:

```

public class DataBaseOperations
{
    /// <exception cref="MasterFileFormatException"></exception>
    /// <exception cref="MasterFileLockedOpenException"></exception>
    public static void ReadRecord(int flag) {
        if (flag == 1)
            throw new MasterFileFormatException();
        else if (flag == 2)
            throw new MasterFileLockedOpenException();
        // ...
    }
}

```

`<include>`

This tag allows including information from an XML document that is external to the source code file. The external file must be a well-formed XML document, and an XPath expression is applied to that document to specify what XML from that document to include. The `<include>` tag is then replaced with the selected XML from the external document.

Syntax:

```
<include file="filename" path="xpath" />
```

where

- `filename` is the file name of an external XML file. The file name is interpreted relative to the file that contains the include tag.
- `xpath` is an XPath expression that selects some of the XML in the external XML file.

Example:

If the source code contained a declaration like:

```

/// <include file="docs.xml" path='extradoc/class[@name="IntList"]/*' />
public class IntList { ... }

```

and the external file "docs.xml" had the following contents:

```

<?xml version="1.0"?>
<extradoc>
  <class name="IntList">
    <summary>
      Contains a list of integers.
    </summary>
  </class>
  <class name="StringList">
    <summary>
      Contains a list of integers.
    </summary>
  </class>
</extradoc>

```

then the same documentation is output as if the source code contained:

```

/// <summary>
///     Contains a list of integers.
/// </summary>
public class IntList { ... }

```

<list>

This tag is used to create a list or table of items. It may contain a <listheader> block to define the heading row of either a table or definition list. (When defining a table, only an entry for <term> in the heading need be supplied.)

Each item in the list is specified with an <item> block. When creating a definition list, both <term> and <description> must be specified. However, for a table, bulleted list, or numbered list, only <description> need be specified.

Syntax:

```

<list type="bullet" | "number" | "table">
  <listheader>
    <term>term</term>
    <description>*description*</description>
  </listheader>
  <item>
    <term>term</term>
    <description>*description*</description>
  </item>
  ...
  <item>
    <term>term</term>
    <description>description</description>
  </item>
</list>

```

where

- <term> is the term to define, whose definition is in <description> .
- <description> is either an item in a bullet or numbered list, or the definition of a <term> .

Example:

```

public class MyClass
{
    /// <summary>Here is an example of a bulleted list:
    /// <list type="bullet">
    /// <item>
    /// <description>Item 1.</description>
    /// </item>
    /// <item>
    /// <description>Item 2.</description>
    /// </item>
    /// </list>
    /// </summary>
    public static void Main () {
        // ...
    }
}

```

<para>

This tag is for use inside other tags, such as <summary> (<remarks>) or <returns> (<returns>), and permits

structure to be added to text.

Syntax:

```
<para>content</para>
```

where `content` is the text of the paragraph.

Example:

```
/// <summary>This is the entry point of the Point class testing program.
/// <para>This program tests each method and operator, and
/// is intended to be run after any non-trivial maintenance has
/// been performed on the Point class.</para></summary>
public static void Main() {
    // ...
}
```

```
<param>
```

This tag is used to describe a parameter for a method, constructor, or indexer.

Syntax:

```
<param name="name">description</param>
```

where

- `name` is the name of the parameter.
- `description` is a description of the parameter.

Example:

```
/// <summary>This method changes the point's location to
/// the given coordinates.</summary>
/// <param name="xor">the new x-coordinate.</param>
/// <param name="yor">the new y-coordinate.</param>
public void Move(int xor, int yor) {
    X = xor;
    Y = yor;
}
```

```
<paramref>
```

This tag is used to indicate that a word is a parameter. The documentation file can be processed to format this parameter in some distinct way.

Syntax:

```
<paramref name="name"/>
```

where `name` is the name of the parameter.

Example:

```

/// <summary>This constructor initializes the new Point to
///     (<paramref name="xor"/>,<paramref name="yor"/>).</summary>
/// <param name="xor">the new Point's x-coordinate.</param>
/// <param name="yor">the new Point's y-coordinate.</param>

public Point(int xor, int yor) {
    X = xor;
    Y = yor;
}

```

`<permission>`

This tag allows the security accessibility of a member to be documented.

Syntax:

```
<permission cref="member">description</permission>
```

where

- `member` is the name of a member. The documentation generator checks that the given code element exists and translates *member* to the canonical element name in the documentation file.
- `description` is a description of the access to the member.

Example:

```

/// <permission cref="System.Security.PermissionSet">Everyone can
/// access this method.</permission>

public static void Test() {
    // ...
}

```

`<remarks>`

This tag is used to specify extra information about a type. (Use `<summary>` (`<summary>`) to describe the type itself and the members of a type.)

Syntax:

```
<remarks>description</remarks>
```

where `description` is the text of the remark.

Example:

```

/// <summary>Class <c>Point</c> models a point in a
/// two-dimensional plane.</summary>
/// <remarks>Uses polar coordinates</remarks>
public class Point
{
    // ...
}

```

`<returns>`

This tag is used to describe the return value of a method.

Syntax:

```
<returns>description</returns>
```

where `description` is a description of the return value.

Example:

```
/// <summary>Report a point's location as a string.</summary>
/// <returns>A string representing a point's location, in the form (x,y),
///     without any leading, trailing, or embedded whitespace.</returns>
public override string ToString() {
    return "(" + X + ", " + Y + ")";
}
```

```
<see>
```

This tag allows a link to be specified within text. Use `<seealso>` (`<seealso>`) to indicate text that is to appear in a See Also section.

Syntax:

```
<see cref="member"/>
```

where `member` is the name of a member. The documentation generator checks that the given code element exists and changes *member* to the element name in the generated documentation file.

Example:

```
/// <summary>This method changes the point's location to
///     the given coordinates.</summary>
/// <see cref="Translate"/>
public void Move(int xor, int yor) {
    X = xor;
    Y = yor;
}

/// <summary>This method changes the point's location by
///     the given x- and y-offsets.
/// </summary>
/// <see cref="Move"/>
public void Translate(int xor, int yor) {
    X += xor;
    Y += yor;
}
```

```
<seealso>
```

This tag allows an entry to be generated for the See Also section. Use `<see>` (`<see>`) to specify a link from within text.

Syntax:

```
<seealso cref="member"/>
```

where `member` is the name of a member. The documentation generator checks that the given code element exists and changes *member* to the element name in the generated documentation file.

Example:

```
/// <summary>This method determines whether two Points have the same
///     location.</summary>
/// <seealso cref="operator==">
/// <seealso cref="operator!=">
public override bool Equals(object o) {
    // ...
}
```

`<summary>`

This tag can be used to describe a type or a member of a type. Use `<remarks>` (`<remarks>`) to describe the type itself.

Syntax:

```
<summary>description</summary>
```

where `description` is a summary of the type or member.

Example:

```
/// <summary>This constructor initializes the new Point to (0,0).</summary>
public Point() : this(0,0) {
}
```

`<value>`

This tag allows a property to be described.

Syntax:

```
<value>property description</value>
```

where `property description` is a description for the property.

Example:

```
/// <value>Property <c>X</c> represents the point's x-coordinate.</value>
public int X
{
    get { return x; }
    set { x = value; }
}
```

`<typeparam>`

This tag is used to describe a generic type parameter for a class, struct, interface, delegate, or method.

Syntax:

```
<typeparam name="name">description</typeparam>
```

where `name` is the name of the type parameter, and `description` is its description.

Example:


```
/// <summary>A generic list class.</summary>
/// <typeparam name="T">The type stored by the list.</typeparam>
public class MyList<T> {
    ...
}
```

`<typeparamref>`

This tag is used to indicate that a word is a type parameter. The documentation file can be processed to format this type parameter in some distinct way.

Syntax:

```
<typeparamref name="name"/>
```

where `name` is the name of the type parameter.

Example:

```
/// <summary>This method fetches data and returns a list of <typeparamref name="T"/>.</summary>
/// <param name="query">query to execute</param>
public List<T> FetchData<T>(string query) {
    ...
}
```

Processing the documentation file

The documentation generator generates an ID string for each element in the source code that is tagged with a documentation comment. This ID string uniquely identifies a source element. A documentation viewer can use an ID string to identify the corresponding metadata/reflection item to which the documentation applies.

The documentation file is not a hierarchical representation of the source code; rather, it is a flat list with a generated ID string for each element.

ID string format

The documentation generator observes the following rules when it generates the ID strings:

- No white space is placed in the string.
- The first part of the string identifies the kind of member being documented, via a single character followed by a colon. The following kinds of members are defined:

CHARACTER	DESCRIPTION
E	Event
F	Field
M	Method (including constructors, destructors, and operators)
N	Namespace
P	Property (including indexers)

CHARACTER	DESCRIPTION
T	Type (such as class, delegate, enum, interface, and struct)
!	Error string; the rest of the string provides information about the error. For example, the documentation generator generates error information for links that cannot be resolved.

- The second part of the string is the fully qualified name of the element, starting at the root of the namespace. The name of the element, its enclosing type(s), and namespace are separated by periods. If the name of the item itself has periods, they are replaced by `#(U+0023)` characters. (It is assumed that no element has this character in its name.)
- For methods and properties with arguments, the argument list follows, enclosed in parentheses. For those without arguments, the parentheses are omitted. The arguments are separated by commas. The encoding of each argument is the same as a CLI signature, as follows:
 - Arguments are represented by their documentation name, which is based on their fully qualified name, modified as follows:
 - Arguments that represent generic types have an appended ``` (backtick) character followed by the number of type parameters
 - Arguments having the `out` or `ref` modifier have an `@` following their type name. Arguments passed by value or via `params` have no special notation.
 - Arguments that are arrays are represented as `[lowerbound:size, ... , lowerbound:size]` where the number of commas is the rank less one, and the lower bounds and size of each dimension, if known, are represented in decimal. If a lower bound or size is not specified, it is omitted. If the lower bound and size for a particular dimension are omitted, the `:` is omitted as well. Jagged arrays are represented by one `[]` per level.
 - Arguments that have pointer types other than void are represented using a `*` following the type name. A void pointer is represented using a type name of `System.Void`.
 - Arguments that refer to generic type parameters defined on types are encoded using the ``` (backtick) character followed by the zero-based index of the type parameter.
 - Arguments that use generic type parameters defined in methods use a double-backtick ```` instead of the ``` used for types.
 - Arguments that refer to constructed generic types are encoded using the generic type, followed by `{`, followed by a comma-separated list of type arguments, followed by `}`.

ID string examples

The following examples each show a fragment of C# code, along with the ID string produced from each source element capable of having a documentation comment:

- Types are represented using their fully qualified name, augmented with generic information:

```

enum Color { Red, Blue, Green }

namespace Acme
{
    interface IProcess {...}

    struct ValueType {...}

    class Widget: IProcess
    {
        public class NestedClass {...}
        public interface IMenuItem {...}
        public delegate void Del(int i);
        public enum Direction { North, South, East, West }
    }

    class MyList<T>
    {
        class Helper<U,V> {...}
    }
}

"T:Color"
"T:Acme.IProcess"
"T:Acme.ValueType"
"T:Acme.Widget"
"T:Acme.Widget.NestedClass"
"T:Acme.Widget.IMenuItem"
"T:Acme.Widget.Del"
"T:Acme.Widget.Direction"
"T:Acme.MyList`1"
"T:Acme.MyList`1.Helper`2"

```

- Fields are represented by their fully qualified name:

```

namespace Acme
{
    struct ValueType
    {
        private int total;
    }

    class Widget: IProcess
    {
        public class NestedClass
        {
            private int value;
        }

        private string message;
        private static Color defaultColor;
        private const double PI = 3.14159;
        protected readonly double monthlyAverage;
        private long[] array1;
        private Widget[,] array2;
        private unsafe int *pCount;
        private unsafe float **ppValues;
    }
}

"F:Acme.ValueType.total"
"F:Acme.Widget.NestedClass.value"
"F:Acme.Widget.message"
"F:Acme.Widget.defaultColor"
"F:Acme.Widget.PI"
"F:Acme.Widget.monthlyAverage"
"F:Acme.Widget.array1"
"F:Acme.Widget.array2"
"F:Acme.Widget.pCount"
"F:Acme.Widget.ppValues"

```

- Constructors.

```

namespace Acme
{
    class Widget: IProcess
    {
        static Widget() {...}
        public Widget() {...}
        public Widget(string s) {...}
    }
}

"M:Acme.Widget.#cctor"
"M:Acme.Widget.#ctor"
"M:Acme.Widget.#ctor(System.String)"

```

- Destructors.

```

namespace Acme
{
    class Widget: IProcess
    {
        ~Widget() {...}
    }
}

"M:Acme.Widget.Finalize"

```

- Methods.

```
namespace Acme
{
    struct ValueType
    {
        public void M(int i) {...}
    }

    class Widget: IProcess
    {
        public class NestedClass
        {
            public void M(int i) {...}
        }

        public static void M0() {...}
        public void M1(char c, out float f, ref ValueType v) {...}
        public void M2(short[] x1, int[,] x2, long[][] x3) {...}
        public void M3(long[][] x3, Widget[,] x4) {...}
        public unsafe void M4(char *pc, Color **pf) {...}
        public unsafe void M5(void *pv, double *[,] pd) {...}
        public void M6(int i, params object[] args) {...}
    }

    class MyList<T>
    {
        public void Test(T t) { }
    }

    class UseList
    {
        public void Process(MyList<int> list) { }
        public MyList<T> GetValues<T>(T inputValue) { return null; }
    }
}

"M:Acme.ValueType.M(System.Int32)"
"M:Acme.Widget.NestedClass.M(System.Int32)"
"M:Acme.Widget.M0"
"M:Acme.Widget.M1(System.Char,System.Single@,Acme.ValueType@)"
"M:Acme.Widget.M2(System.Int16[],System.Int32[0:,0:],System.Int64[][])"
"M:Acme.Widget.M3(System.Int64[][],Acme.Widget[0:,0:,0:][])"
"M:Acme.Widget.M4(System.Char*,Color**)"
"M:Acme.Widget.M5(System.Void*,System.Double*[0:,0:][])"
"M:Acme.Widget.M6(System.Int32,System.Object[])"
"M:Acme.MyList`1.Test(`0)"
"M:Acme.UseList.Process(Acme.MyList{System.Int32})"
"M:Acme.UseList.GetValues``(``0)"
```

- Properties and indexers.

```

namespace Acme
{
    class Widget: IProcess
    {
        public int Width { get {...} set {...} }
        public int this[int i] { get {...} set {...} }
        public int this[string s, int i] { get {...} set {...} }
    }
}

"P:Acme.Widget.Width"
"P:Acme.Widget.Item(System.Int32)"
"P:Acme.Widget.Item(System.String,System.Int32)"

```

- Events.

```

namespace Acme
{
    class Widget: IProcess
    {
        public event Del AnEvent;
    }
}

"E:Acme.Widget.AnEvent"

```

- Unary operators.

```

namespace Acme
{
    class Widget: IProcess
    {
        public static Widget operator+(Widget x) {...}
    }
}

"M:Acme.Widget.op_UnaryPlus(Acme.Widget)"

```

The complete set of unary operator function names used is as follows: `op_UnaryPlus`, `op_UnaryNegation`, `op_LogicalNot`, `op_OnesComplement`, `op_Increment`, `op_Decrement`, `op_True`, and `op_False`.

- Binary operators.

```

namespace Acme
{
    class Widget: IProcess
    {
        public static Widget operator+(Widget x1, Widget x2) {...}
    }
}

"M:Acme.Widget.op_Addition(Acme.Widget,Acme.Widget)"

```

The complete set of binary operator function names used is as follows: `op_Addition`, `op_Subtraction`, `op_Multiply`, `op_Division`, `op_Modulus`, `op_BitwiseAnd`, `op_BitwiseOr`, `op_ExclusiveOr`, `op_LeftShift`, `op_RightShift`, `op_Equality`, `op_Inequality`, `op_LessThan`, `op_LessThanOrEqual`, `op_GreaterThan`, and `op_GreaterThanOrEqual`.

- Conversion operators have a trailing "`~`" followed by the return type.

```

namespace Acme
{
    class Widget: IProcess
    {
        public static explicit operator int(Widget x) {...}
        public static implicit operator long(Widget x) {...}
    }
}

"M:Acme.Widget.op_Explicit(Acme.Widget)~System.Int32"
"M:Acme.Widget.op_Implicit(Acme.Widget)~System.Int64"

```

An example

C# source code

The following example shows the source code of a `Point` class:

```

namespace Graphics
{
    /// <summary>Class <c>Point</c> models a point in a two-dimensional plane.
    /// </summary>
    public class Point
    {
        /// <summary>Instance variable <c>x</c> represents the point's
        ///     x-coordinate.</summary>
        private int x;

        /// <summary>Instance variable <c>y</c> represents the point's
        ///     y-coordinate.</summary>
        private int y;

        /// <value>Property <c>X</c> represents the point's x-coordinate.</value>
        public int X
        {
            get { return x; }
            set { x = value; }
        }

        /// <value>Property <c>Y</c> represents the point's y-coordinate.</value>
        public int Y
        {
            get { return y; }
            set { y = value; }
        }

        /// <summary>This constructor initializes the new Point to
        ///     (0,0).</summary>
        public Point() : this(0,0) {}

        /// <summary>This constructor initializes the new Point to
        ///     (<paramref name="xor"/>,<paramref name="yor"/>).</summary>
        /// <param><c>xor</c> is the new Point's x-coordinate.</param>
        /// <param><c>yor</c> is the new Point's y-coordinate.</param>
        public Point(int xor, int yor) {
            X = xor;
            Y = yor;
        }

        /// <summary>This method changes the point's location to
        ///     the given coordinates.</summary>
        /// <param><c>xor</c> is the new x-coordinate.</param>
        /// <param><c>yor</c> is the new y-coordinate.</param>
    }
}

```

```

/// <see cref="Translate"/>
public void Move(int xor, int yor) {
    X = xor;
    Y = yor;
}

/// <summary>This method changes the point's location by
///     the given x- and y-offsets.
/// <example>For example:
/// <code>
///     Point p = new Point(3,5);
///     p.Translate(-1,3);
/// </code>
/// results in <c>p</c>'s having the value (2,8).
/// </example>
/// </summary>
/// <param><c>xor</c> is the relative x-offset.</param>
/// <param><c>yor</c> is the relative y-offset.</param>
/// <see cref="Move"/>
public void Translate(int xor, int yor) {
    X += xor;
    Y += yor;
}

/// <summary>This method determines whether two Points have the same
///     location.</summary>
/// <param><c>o</c> is the object to be compared to the current object.
/// </param>
/// <returns>True if the Points have the same location and they have
///     the exact same type; otherwise, false.</returns>
/// <seealso cref="operator==">
/// <seealso cref="operator!=">
public override bool Equals(object o) {
    if (o == null) {
        return false;
    }

    if (this == o) {
        return true;
    }

    if (GetType() == o.GetType()) {
        Point p = (Point)o;
        return (X == p.X) && (Y == p.Y);
    }

    return false;
}

/// <summary>Report a point's location as a string.</summary>
/// <returns>A string representing a point's location, in the form (x,y),
///     without any leading, training, or embedded whitespace.</returns>
public override string ToString() {
    return "(" + X + ", " + Y + ")";
}

/// <summary>This operator determines whether two Points have the same
///     location.</summary>
/// <param><c>p1</c> is the first Point to be compared.</param>
/// <param><c>p2</c> is the second Point to be compared.</param>
/// <returns>True if the Points have the same location and they have
///     the exact same type; otherwise, false.</returns>
/// <seealso cref="Equals"/>
/// <seealso cref="operator!=">
public static bool operator==(Point p1, Point p2) {
    if ((object)p1 == null || (object)p2 == null) {
        return false;
    }

    if (p1.GetType() == p2.GetType()) {

```



```

        return (p1.X == p2.X) && (p1.Y == p2.Y);
    }

    return false;
}

/// <summary>This operator determines whether two Points have the same
///     location.</summary>
/// <param><c>p1</c> is the first Point to be compared.</param>
/// <param><c>p2</c> is the second Point to be compared.</param>
/// <returns>True if the Points do not have the same location and the
///     exact same type; otherwise, false.</returns>
/// <seealso cref="Equals"/>
/// <seealso cref="operator==">
public static bool operator!=(Point p1, Point p2) {
    return !(p1 == p2);
}

/// <summary>This is the entry point of the Point class testing
/// program.
/// <para>This program tests each method and operator, and
/// is intended to be run after any non-trivial maintenance has
/// been performed on the Point class.</para></summary>
public static void Main() {
    // class test code goes here
}
}
}

```

Resulting XML

Here is the output produced by one documentation generator when given the source code for class `Point`, shown above:

```

<?xml version="1.0"?>
<doc>
  <assembly>
    <name>Point</name>
  </assembly>
  <members>
    <member name="T:Graphics.Point">
      <summary>Class <c>Point</c> models a point in a two-dimensional
        plane.
      </summary>
    </member>

    <member name="F:Graphics.Point.x">
      <summary>Instance variable <c>x</c> represents the point's
        x-coordinate.</summary>
    </member>

    <member name="F:Graphics.Point.y">
      <summary>Instance variable <c>y</c> represents the point's
        y-coordinate.</summary>
    </member>

    <member name="M:Graphics.Point.#ctor">
      <summary>This constructor initializes the new Point to
        (0,0).</summary>
    </member>

    <member name="M:Graphics.Point.#ctor(System.Int32,System.Int32)">
      <summary>This constructor initializes the new Point to
        (<paramref name="xor"/>,<paramref name="yor"/>).</summary>
      <param><c>xor</c> is the new Point's x-coordinate.</param>
      <param><c>yor</c> is the new Point's y-coordinate.</param>
    </member>
  </members>
</doc>

```

```

<member name="M:Graphics.Point.Move(System.Int32,System.Int32)">
    <summary>This method changes the point's location to
    the given coordinates.</summary>
    <param><c>xor</c> is the new x-coordinate.</param>
    <param><c>yor</c> is the new y-coordinate.</param>
    <see cref="M:Graphics.Point.Translate(System.Int32,System.Int32)"/>
</member>

<member
    name="M:Graphics.Point.Translate(System.Int32,System.Int32)">
    <summary>This method changes the point's location by
    the given x- and y-offsets.
    <example>For example:
    <code>
    Point p = new Point(3,5);
    p.Translate(-1,3);
    </code>
    results in <c>p</c>'s having the value (2,8).
    </example>
    </summary>
    <param><c>xor</c> is the relative x-offset.</param>
    <param><c>yor</c> is the relative y-offset.</param>
    <see cref="M:Graphics.Point.Move(System.Int32,System.Int32)"/>
</member>

<member name="M:Graphics.Point.Equals(System.Object)">
    <summary>This method determines whether two Points have the same
    location.</summary>
    <param><c>o</c> is the object to be compared to the current
    object.
    </param>
    <returns>True if the Points have the same location and they have
    the exact same type; otherwise, false.</returns>
    <seealso
    cref="M:Graphics.Point.op_Equality(Graphics.Point,Graphics.Point)"/>
    <seealso
    cref="M:Graphics.Point.op_Inequality(Graphics.Point,Graphics.Point)"/>
</member>

<member name="M:Graphics.Point.ToString">
    <summary>Report a point's location as a string.</summary>
    <returns>A string representing a point's location, in the form
    (x,y),
    without any leading, training, or embedded whitespace.</returns>
</member>

<member
    name="M:Graphics.Point.op_Equality(Graphics.Point,Graphics.Point)">
    <summary>This operator determines whether two Points have the
    same
    location.</summary>
    <param><c>p1</c> is the first Point to be compared.</param>
    <param><c>p2</c> is the second Point to be compared.</param>
    <returns>True if the Points have the same location and they have
    the exact same type; otherwise, false.</returns>
    <seealso cref="M:Graphics.Point.Equals(System.Object)"/>
    <seealso
    cref="M:Graphics.Point.op_Inequality(Graphics.Point,Graphics.Point)"/>
</member>

<member
    name="M:Graphics.Point.op_Inequality(Graphics.Point,Graphics.Point)">
    <summary>This operator determines whether two Points have the
    same
    location.</summary>
    <param><c>p1</c> is the first Point to be compared.</param>
    <param><c>p2</c> is the second Point to be compared.</param>
    <returns>True if the Points do not have the same location and

```

```
        the
        exact same type; otherwise, false.</returns>
        <seealso cref="M:Graphics.Point.Equals(System.Object)"/>
        <seealso
cref="M:Graphics.Point.op_Equality(Graphics.Point,Graphics.Point)"/>
    </member>

    <member name="M:Graphics.Point.Main">
        <summary>This is the entry point of the Point class testing
        program.
        <para>This program tests each method and operator, and
        is intended to be run after any non-trivial maintenance has
        been performed on the Point class.</para></summary>
    </member>

    <member name="P:Graphics.Point.X">
        <value>Property <c>X</c> represents the point's
        x-coordinate.</value>
    </member>

    <member name="P:Graphics.Point.Y">
        <value>Property <c>Y</c> represents the point's
        y-coordinate.</value>
    </member>
</members>
</doc>
```

Pattern Matching for C# 7

12/28/2021 • 15 minutes to read • [Edit Online](#)

Pattern matching extensions for C# enable many of the benefits of algebraic data types and pattern matching from functional languages, but in a way that smoothly integrates with the feel of the underlying language. The basic features are: [record types](#), which are types whose semantic meaning is described by the shape of the data; and pattern matching, which is a new expression form that enables extremely concise multilevel decomposition of these data types. Elements of this approach are inspired by related features in the programming languages [F#](#) and [Scala](#).

Is expression

The `is` operator is extended to test an expression against a *pattern*.

```
relational_expression
    : relational_expression 'is' pattern
    ;
```

This form of *relational_expression* is in addition to the existing forms in the C# specification. It is a compile-time error if the *relational_expression* to the left of the `is` token does not designate a value or does not have a type.

Every *identifier* of the pattern introduces a new local variable that is *definitely assigned* after the `is` operator is `true` (i.e. *definitely assigned when true*).

Note: There is technically an ambiguity between *type* in an `is-expression` and *constant_pattern*, either of which might be a valid parse of a qualified identifier. We try to bind it as a type for compatibility with previous versions of the language; only if that fails do we resolve it as we do in other contexts, to the first thing found (which must be either a constant or a type). This ambiguity is only present on the right-hand-side of an `is` expression.

Patterns

Patterns are used in the `is` operator and in a *switch_statement* to express the shape of data against which incoming data is to be compared. Patterns may be recursive so that parts of the data may be matched against sub-patterns.

```

pattern
    : declaration_pattern
    | constant_pattern
    | var_pattern
    ;

declaration_pattern
    : type simple_designation
    ;

constant_pattern
    : shift_expression
    ;

var_pattern
    : 'var' simple_designation
    ;

```

Note: There is technically an ambiguity between *type* in an `is-expression` and *constant_pattern*, either of which might be a valid parse of a qualified identifier. We try to bind it as a type for compatibility with previous versions of the language; only if that fails do we resolve it as we do in other contexts, to the first thing found (which must be either a constant or a type). This ambiguity is only present on the right-hand-side of an `is` expression.

Declaration pattern

The *declaration_pattern* both tests that an expression is of a given type and casts it to that type if the test succeeds. If the *simple_designation* is an identifier, it introduces a local variable of the given type named by the given identifier. That local variable is *definitely assigned* when the result of the pattern-matching operation is true.

```

declaration_pattern
    : type simple_designation
    ;

```

The runtime semantic of this expression is that it tests the runtime type of the left-hand *relational_expression* operand against the *type* in the pattern. If it is of that runtime type (or some subtype), the result of the `is operator` is `true`. It declares a new local variable named by the *identifier* that is assigned the value of the left-hand operand when the result is `true`.

Certain combinations of static type of the left-hand-side and the given type are considered incompatible and result in compile-time error. A value of static type `E` is said to be *pattern compatible* with the type `T` if there exists an identity conversion, an implicit reference conversion, a boxing conversion, an explicit reference conversion, or an unboxing conversion from `E` to `T`. It is a compile-time error if an expression of type `E` is not pattern compatible with the type in a type pattern that it is matched with.

Note: In C# 7.1 we extend this to permit a pattern-matching operation if either the input type or the type `T` is an open type. This paragraph is replaced by the following:

Certain combinations of static type of the left-hand-side and the given type are considered incompatible and result in compile-time error. A value of static type `E` is said to be *pattern compatible* with the type `T` if there exists an identity conversion, an implicit reference conversion, a boxing conversion, an explicit reference conversion, or an unboxing conversion from `E` to `T`, or if either `E` or `T` is an open type. It is a compile-time error if an expression of type `E` is not pattern compatible with the type in a type pattern that it is matched with.

The declaration pattern is useful for performing run-time type tests of reference types, and replaces the idiom

```
var v = expr as Type;
if (v != null) { // code using v }
```

With the slightly more concise

```
if (expr is Type v) { // code using v }
```

It is an error if *type* is a nullable value type.

The declaration pattern can be used to test values of nullable types: a value of type `Nullable<T>` (or a boxed `T`) matches a type pattern `T2 id` if the value is non-null and the type of `T2` is `T`, or some base type or interface of `T`. For example, in the code fragment

```
int? x = 3;
if (x is int v) { // code using v }
```

The condition of the `if` statement is `true` at runtime and the variable `v` holds the value `3` of type `int` inside the block.

Constant pattern

```
constant_pattern
: shift_expression
;
```

A constant pattern tests the value of an expression against a constant value. The constant may be any constant expression, such as a literal, the name of a declared `const` variable, or an enumeration constant, or a `typeof` expression.

If both *e* and *c* are of integral types, the pattern is considered matched if the result of the expression `e == c` is `true`.

Otherwise the pattern is considered matching if `object.Equals(e, c)` returns `true`. In this case it is a compile-time error if the static type of *e* is not *pattern compatible* with the type of the constant.

Var pattern

```
var_pattern
: 'var' simple_designation
;
```

An expression *e* matches a *var_pattern* always. In other words, a match to a *var pattern* always succeeds. If the *simple_designation* is an identifier, then at runtime the value of *e* is bound to a newly introduced local variable. The type of the local variable is the static type of *e*.

It is an error if the name `var` binds to a type.

Switch statement

The `switch` statement is extended to select for execution the first block having an associated pattern that matches the *switch expression*.

```

switch_label
  : 'case' complex_pattern case_guard? ':'
  | 'case' constant_expression case_guard? ':'
  | 'default' ':'
  ;

case_guard
  : 'when' expression
  ;

```

The order in which patterns are matched is not defined. A compiler is permitted to match patterns out of order, and to reuse the results of already matched patterns to compute the result of matching of other patterns.

If a *case-guard* is present, its expression is of type `bool`. It is evaluated as an additional condition that must be satisfied for the case to be considered satisfied.

It is an error if a *switch_label* can have no effect at runtime because its pattern is subsumed by previous cases. [TODO: We should be more precise about the techniques the compiler is required to use to reach this judgment.]

A pattern variable declared in a *switch_label* is definitely assigned in its case block if and only if that case block contains precisely one *switch_label*.

[TODO: We should specify when a *switch block* is reachable.]

Scope of pattern variables

The scope of a variable declared in a pattern is as follows:

- If the pattern is a case label, then the scope of the variable is the *case block*.

Otherwise the variable is declared in an *is_pattern* expression, and its scope is based on the construct immediately enclosing the expression containing the *is_pattern* expression as follows:

- If the expression is in an expression-bodied lambda, its scope is the body of the lambda.
- If the expression is in an expression-bodied method or property, its scope is the body of the method or property.
- If the expression is in a `when` clause of a `catch` clause, its scope is that `catch` clause.
- If the expression is in an *iteration_statement*, its scope is just that statement.
- Otherwise if the expression is in some other statement form, its scope is the scope containing the statement.

For the purpose of determining the scope, an *embedded_statement* is considered to be in its own scope. For example, the grammar for an *if_statement* is

```

if_statement
  : 'if' '(' boolean_expression ')' embedded_statement
  | 'if' '(' boolean_expression ')' embedded_statement 'else' embedded_statement
  ;

```

So if the controlled statement of an *if_statement* declares a pattern variable, its scope is restricted to that *embedded_statement*.

```
if (x) M(y is var z);
```

In this case the scope of `z` is the embedded statement `M(y is var z);`.

Other cases are errors for other reasons (e.g. in a parameter's default value or an attribute, both of which are an error because those contexts require a constant expression).

In C# 7.3 we added the following contexts in which a pattern variable may be declared:

- If the expression is in a *constructor initializer*, its scope is the *constructor initializer* and the constructor's body.
- If the expression is in a field initializer, its scope is the *equals_value_clause* in which it appears.
- If the expression is in a query clause that is specified to be translated into the body of a lambda, its scope is just that expression.

Changes to syntactic disambiguation

There are situations involving generics where the C# grammar is ambiguous, and the language spec says how to resolve those ambiguities:

7.6.5.2 Grammar ambiguities

The productions for *simple-name* (§7.6.3) and *member-access* (§7.6.5) can give rise to ambiguities in the grammar for expressions. For example, the statement:

```
F(G<A,B>(7));
```

could be interpreted as a call to `F` with two arguments, `G < A` and `B > (7)`. Alternatively, it could be interpreted as a call to `F` with one argument, which is a call to a generic method `G` with two type arguments and one regular argument.

If a sequence of tokens can be parsed (in context) as a *simple-name* (§7.6.3), *member-access* (§7.6.5), or *pointer-member-access* (§18.5.2) ending with a *type-argument-list* (§4.4.1), the token immediately following the closing `>` token is examined. If it is one of

```
( ) ] } : ; , . ? == != | ^
```

then the *type-argument-list* is retained as part of the *simple-name*, *member-access* or *pointer-member-access* and any other possible parse of the sequence of tokens is discarded. Otherwise, the *type-argument-list* is not considered to be part of the *simple-name*, *member-access* or *> pointer-member-access*, even if there is no other possible parse of the sequence of tokens. Note that these rules are not applied when parsing a *type-argument-list* in a *namespace-or-type-name* (§3.8). The statement

```
F(G<A,B>(7));
```

will, according to this rule, be interpreted as a call to `F` with one argument, which is a call to a generic method `G` with two type arguments and one regular argument. The statements

```
F(G < A, B > 7);  
F(G < A, B >> 7);
```

will each be interpreted as a call to `F` with two arguments. The statement

```
x = F < A > +y;
```

will be interpreted as a less than operator, greater than operator, and unary plus operator, as if the statement had been written `x = (F < A) > (+y)`, instead of as a *simple-name* with a *type-argument-list* followed by a binary plus operator. In the statement


```
x = y is C<T> + z;
```

the tokens `C<T>` are interpreted as a *namespace-or-type-name* with a *type-argument-list*.

There are a number of changes being introduced in C# 7 that make these disambiguation rules no longer sufficient to handle the complexity of the language.

Out variable declarations

It is now possible to declare a variable in an out argument:

```
M(out Type name);
```

However, the type may be generic:

```
M(out A<B> name);
```

Since the language grammar for the argument uses *expression*, this context is subject to the disambiguation rule. In this case the closing `>` is followed by an *identifier*, which is not one of the tokens that permits it to be treated as a *type-argument-list*. I therefore propose to **add *identifier* to the set of tokens that triggers the disambiguation to a *type-argument-list*.**

Tuples and deconstruction declarations

A tuple literal runs into exactly the same issue. Consider the tuple expression

```
(A < B, C > D, E < F, G > H)
```

Under the old C# 6 rules for parsing an argument list, this would parse as a tuple with four elements, starting with `A < B` as the first. However, when this appears on the left of a deconstruction, we want the disambiguation triggered by the *identifier* token as described above:

```
(A<B,C> D, E<F,G> H) = e;
```

This is a deconstruction declaration which declares two variables, the first of which is of type `A<B,C>` and named `D`. In other words, the tuple literal contains two expressions, each of which is a declaration expression.

For simplicity of the specification and compiler, I propose that this tuple literal be parsed as a two-element tuple wherever it appears (whether or not it appears on the left-hand-side of an assignment). That would be a natural result of the disambiguation described in the previous section.

Pattern-matching

Pattern matching introduces a new context where the expression-type ambiguity arises. Previously the right-hand-side of an `is` operator was a type. Now it can be a type or expression, and if it is a type it may be followed by an identifier. This can, technically, change the meaning of existing code:

```
var x = e is T < A > B;
```

This could be parsed under C#6 rules as

```
var x = ((e is T) < A) > B;
```

but under C#7 rules (with the disambiguation proposed above) would be parsed as

```
var x = e is T<A> B;
```

which declares a variable `B` of type `T<A>`. Fortunately, the native and Roslyn compilers have a bug whereby they give a syntax error on the C#6 code. Therefore this particular breaking change is not a concern.

Pattern-matching introduces additional tokens that should drive the ambiguity resolution toward selecting a type. The following examples of existing valid C#6 code would be broken without additional disambiguation rules:

```
var x = e is A<B> && f;           // &&
var x = e is A<B> || f;           // ||
var x = e is A<B> & f;            // &
var x = e is A<B>[];              // [
```

Proposed change to the disambiguation rule

I propose to revise the specification to change the list of disambiguating tokens from

```
( ) ] } : ; , . ? == != | ^
```

to

```
( ) ] } : ; , . ? == != | ^ && || & [
```

And, in certain contexts, we treat *identifier* as a disambiguating token. Those contexts are where the sequence of tokens being disambiguated is immediately preceded by one of the keywords `is`, `case`, or `out`, or arises while parsing the first element of a tuple literal (in which case the tokens are preceded by `(` or `:` and the identifier is followed by a `,`) or a subsequent element of a tuple literal.

Modified disambiguation rule

The revised disambiguation rule would be something like this

If a sequence of tokens can be parsed (in context) as a *simple-name* (§7.6.3), *member-access* (§7.6.5), or *pointer-member-access* (§18.5.2) ending with a *type-argument-list* (§4.4.1), the token immediately following the closing `>` token is examined, to see if it is

- One of `()] } : ; , . ? == != | ^ && || & [`; or
- One of the relational operators `< > <= >= is as`; or
- A contextual query keyword appearing inside a query expression; or
- In certain contexts, we treat *identifier* as a disambiguating token. Those contexts are where the sequence of tokens being disambiguated is immediately preceded by one of the keywords `is`, `case` or `out`, or arises while parsing the first element of a tuple literal (in which case the tokens are preceded by `(` or `:` and the identifier is followed by a `,`) or a subsequent element of a tuple literal.

If the following token is among this list, or an identifier in such a context, then the *type-argument-list* is retained as part of the *simple-name*, *member-access* or *pointer-member-access* and any other possible parse of the sequence of tokens is discarded. Otherwise, the *type-argument-list* is not considered to be part of the *simple-name*, *member-access* or *pointer-member-access*, even if there is no other possible parse of the sequence of tokens. Note that these rules are not applied when parsing a *type-argument-list* in a

namespace-or-type-name (§3.8).

Breaking changes due to this proposal

No breaking changes are known due to this proposed disambiguation rule.

Interesting examples

Here are some interesting results of these disambiguation rules:

The expression `(A < B, C > D)` is a tuple with two elements, each a comparison.

The expression `(A<B,C> D, E)` is a tuple with two elements, the first of which is a declaration expression.

The invocation `M(A < B, C > D, E)` has three arguments.

The invocation `M(out A<B,C> D, E)` has two arguments, the first of which is an `out` declaration.

The expression `e is A C` uses a declaration expression.

The case label `case A C:` uses a declaration expression.

Some examples of pattern matching

Is-As

We can replace the idiom

```
var v = expr as Type;
if (v != null) {
    // code using v
}
```

With the slightly more concise and direct

```
if (expr is Type v) {
    // code using v
}
```

Testing nullable

We can replace the idiom

```
Type? v = x?.y?.z;
if (v.HasValue) {
    var value = v.GetValueOrDefault();
    // code using value
}
```

With the slightly more concise and direct

```
if (x?.y?.z is Type value) {
    // code using value
}
```

Arithmetic simplification

Suppose we define a set of recursive types to represent expressions (per a separate proposal):

```
abstract class Expr;
class X() : Expr;
class Const(double Value) : Expr;
class Add(Expr Left, Expr Right) : Expr;
class Mult(Expr Left, Expr Right) : Expr;
class Neg(Expr Value) : Expr;
```

Now we can define a function to compute the (unreduced) derivative of an expression:

```
Expr Deriv(Expr e)
{
    switch (e) {
        case X(): return Const(1);
        case Const(*): return Const(0);
        case Add(var Left, var Right):
            return Add(Deriv(Left), Deriv(Right));
        case Mult(var Left, var Right):
            return Add(Mult(Deriv(Left), Right), Mult(Left, Deriv(Right)));
        case Neg(var Value):
            return Neg(Deriv(Value));
    }
}
```

An expression simplifier demonstrates positional patterns:

```
Expr Simplify(Expr e)
{
    switch (e) {
        case Mult(Const(0), *): return Const(0);
        case Mult(*, Const(0)): return Const(0);
        case Mult(Const(1), var x): return Simplify(x);
        case Mult(var x, Const(1)): return Simplify(x);
        case Mult(Const(var l), Const(var r)): return Const(l*r);
        case Add(Const(0), var x): return Simplify(x);
        case Add(var x, Const(0)): return Simplify(x);
        case Add(Const(var l), Const(var r)): return Const(l+r);
        case Neg(Const(var k)): return Const(-k);
        default: return e;
    }
}
```

Local functions

12/28/2021 • 2 minutes to read • [Edit Online](#)

We extend C# to support the declaration of functions in block scope. Local functions may use (capture) variables from the enclosing scope.

The compiler uses flow analysis to detect which variables a local function uses before assigning it a value. Every call of the function requires such variables to be definitely assigned. Similarly the compiler determines which variables are definitely assigned on return. Such variables are considered definitely assigned after the local function is invoked.

Local functions may be called from a lexical point before its definition. Local function declaration statements do not cause a warning when they are not reachable.

TODO: *WRITE SPEC*

Syntax grammar

This grammar is represented as a diff from the current spec grammar.

```
declaration-statement
    : local-variable-declaration ';'
    | local-constant-declaration ';'
+   | local-function-declaration
    ;

+local-function-declaration
+   : local-function-header local-function-body
+   ;

+local-function-header
+   : local-function-modifiers? return-type identifier type-parameter-list?
+     ( formal-parameter-list? ) type-parameter-constraints-clauses
+   ;

+local-function-modifiers
+   : (async | unsafe)
+   ;

+local-function-body
+   : block
+   | arrow-expression-body
+   ;
```

Local functions may use variables defined in the enclosing scope. The current implementation requires that every variable read inside a local function be definitely assigned, as if executing the local function at its point of definition. Also, the local function definition must have been "executed" at any use point.

After experimenting with that a bit (for example, it is not possible to define two mutually recursive local functions), we've since revised how we want the definite assignment to work. The revision (not yet implemented) is that all local variables read in a local function must be definitely assigned at each invocation of the local function. That's actually more subtle than it sounds, and there is a bunch of work remaining to make it work. Once it is done you'll be able to move your local functions to the end of its enclosing block.

The new definite assignment rules are incompatible with inferring the return type of a local function, so we'll likely be removing support for inferring the return type.

Unless you convert a local function to a delegate, capturing is done into frames that are value types. That means you don't get any GC pressure from using local functions with capturing.

Reachability

We add to the spec

The body of a statement-bodied lambda expression or local function is considered reachable.

Out variable declarations

12/28/2021 • 2 minutes to read • [Edit Online](#)

The *out variable declaration* feature enables a variable to be declared at the location that it is being passed as an `out` argument.

```
argument_value
: 'out' type identifier
| ...
;
```

A variable declared this way is called an *out variable*. You may use the contextual keyword `var` for the variable's type. The scope will be the same as for a *pattern-variable* introduced via pattern-matching.

According to Language Specification (section 7.6.7 Element access) the argument-list of an element-access (indexing expression) does not contain `ref` or `out` arguments. However, they are permitted by the compiler for various scenarios, for example indexers declared in metadata that accept `out`.

Within the scope of a local variable introduced by an `argument_value`, it is a compile-time error to refer to that local variable in a textual position that precedes its declaration.

It is also an error to reference an implicitly-typed (§8.5.1) `out` variable in the same argument list that immediately contains its declaration.

Overload resolution is modified as follows:

We add a new conversion:

There is a *conversion from expression* from an implicitly-typed `out` variable declaration to every type.

Also

The type of an explicitly-typed `out` variable argument is the declared type.

and

An implicitly-typed `out` variable argument has no type.

The *conversion from expression* from an implicitly-typed `out` variable declaration is not considered better than any other *conversion from expression*.

The type of an implicitly-typed `out` variable is the type of the corresponding parameter in the signature of the method selected by overload resolution.

The new syntax node `DeclarationExpressionSyntax` is added to represent the declaration in an `out var` argument.

Throw expression

12/28/2021 • 2 minutes to read • [Edit Online](#)

We extend the set of expression forms to include

```
throw_expression
: 'throw' null_coalescing_expression
;

null_coalescing_expression
: throw_expression
;
```

The type rules are as follows:

- A *throw_expression* has no type.
- A *throw_expression* is convertible to every type by an implicit conversion.

A *throw expression* throws the value produced by evaluating the *null_coalescing_expression*, which must denote a value of the class type `System.Exception`, of a class type that derives from `System.Exception` or of a type parameter type that has `System.Exception` (or a subclass thereof) as its effective base class. If evaluation of the expression produces `null`, a `System.NullReferenceException` is thrown instead.

The behavior at runtime of the evaluation of a *throw expression* is the same [as specified for a *throw statement*](#).

The flow-analysis rules are as follows:

- For every variable v , v is definitely assigned before the *null_coalescing_expression* of a *throw_expression* iff it is definitely assigned before the *throw_expression*.
- For every variable v , v is definitely assigned after *throw_expression*.

A *throw expression* is permitted in only the following syntactic contexts:

- As the second or third operand of a ternary conditional operator `?:`
- As the second operand of a null coalescing operator `??`
- As the body of an expression-bodied lambda or method.

Binary literals

12/28/2021 • 2 minutes to read • [Edit Online](#)

There's a relatively common request to add binary literals to C# and VB. For bitmasks (e.g. flag enums) this seems genuinely useful, but it would also be great just for educational purposes.

Binary literals would look like this:

```
int nineteen = 0b10011;
```

Syntactically and semantically they are identical to hexadecimal literals, except for using `b` / `B` instead of `x` / `X`, having only digits `0` and `1` and being interpreted in base 2 instead of 16.

There's little cost to implementing these, and little conceptual overhead to users of the language.

Syntax

The grammar would be as follows:

```
integer-literal:
    : ...
    | binary-integer-literal
    ;
binary-integer-literal:
    : `0b` binary-digits integer-type-suffix-opt
    | `0B` binary-digits integer-type-suffix-opt
    ;
binary-digits:
    : binary-digit
    | binary-digits binary-digit
    ;
binary-digit:
    : `0`
    | `1`
    ;
```

Digit separators

12/28/2021 • 2 minutes to read • [Edit Online](#)

Being able to group digits in large numeric literals would have great readability impact and no significant downside.

Adding binary literals (#215) would increase the likelihood of numeric literals being long, so the two features enhance each other.

We would follow Java and others, and use an underscore `_` as a digit separator. It would be able to occur everywhere in a numeric literal (except as the first and last character), since different groupings may make sense in different scenarios and especially for different numeric bases:

```
int bin = 0b1001_1010_0001_0100;  
int hex = 0x1b_a0_44_fe;  
int dec = 33_554_432;  
int weird = 1_2_3_4_5_6_7_8_9;  
double real = 1_000.111_1e-1_000;
```

Any sequence of digits may be separated by underscores, possibly more than one underscore between two consecutive digits. They are allowed in decimals as well as exponents, but following the previous rule, they may not appear next to the decimal (`10_.0`), next to the exponent character (`1.1e_1`), or next to the type specifier (`10_f`). When used in binary and hexadecimal literals, they may not appear immediately following the `0x` or `0b`.

The syntax is straightforward, and the separators have no semantic impact - they are simply ignored.

This has broad value and is easy to implement.

Async Task Types in C#

12/28/2021 • 3 minutes to read • [Edit Online](#)

Extend `async` to support *task types* that match a specific pattern, in addition to the well known types

`System.Threading.Tasks.Task` and `System.Threading.Tasks.Task<T>`.

Task Type

A *task type* is a `class` or `struct` with an associated *builder type* identified with `System.Runtime.CompilerServices.AsyncMethodBuilderAttribute`. The *task type* may be non-generic, for async methods that do not return a value, or generic, for methods that return a value.

To support `await`, the *task type* must have a corresponding, accessible `GetAwaiter()` method that returns an instance of an *awaiter type* (see *C# 7.7.7.1 Awaitable expressions*).

```
[AsyncMethodBuilder(typeof(MyTaskMethodBuilder<>))]  
class MyTask<T>  
{  
    public Awaiter<T> GetAwaiter();  
}  
  
class Awaiter<T> : INotifyCompletion  
{  
    public bool IsCompleted { get; }  
    public T GetResult();  
    public void OnCompleted(Action completion);  
}
```

Builder Type

The *builder type* is a `class` or `struct` that corresponds to the specific *task type*. The *builder type* can have at most 1 type parameter and must not be nested in a generic type. The *builder type* has the following `public` methods. For non-generic *builder types*, `SetResult()` has no parameters.

```

class MyTaskMethodBuilder<T>
{
    public static MyTaskMethodBuilder<T> Create();

    public void Start<TStateMachine>(ref TStateMachine stateMachine)
        where TStateMachine : IAsyncStateMachine;

    public void SetStateMachine(IAsyncStateMachine stateMachine);
    public void SetException(Exception exception);
    public void SetResult(T result);

    public void AwaitOnCompleted<TAwaiter, TStateMachine>(
        ref TAwaiter awaiter, ref TStateMachine stateMachine)
        where TAwaiter : INotifyCompletion
        where TStateMachine : IAsyncStateMachine;
    public void AwaitUnsafeOnCompleted<TAwaiter, TStateMachine>(
        ref TAwaiter awaiter, ref TStateMachine stateMachine)
        where TAwaiter : ICriticalNotifyCompletion
        where TStateMachine : IAsyncStateMachine;

    public MyTask<T> Task { get; }
}

```

Execution

The types above are used by the compiler to generate the code for the state machine of an `async` method. (The generated code is equivalent to the code generated for `async` methods that return `Task`, `Task<T>`, or `void`. The difference is, for those well known types, the *builder types* are also known to the compiler.)

`Builder.Create()` is invoked to create an instance of the *builder type*.

`builder.Start(ref stateMachine)` is invoked to associate the builder with compiler-generated state machine instance. The builder must call `stateMachine.MoveNext()` either in `Start()` or after `Start()` has returned to advance the state machine. After `Start()` returns, the `async` method calls `builder.Task` for the task to return from the `async` method.

Each call to `stateMachine.MoveNext()` will advance the state machine. If the state machine completes successfully, `builder.SetResult()` is called, with the method return value if any. If an exception is thrown in the state machine, `builder.SetException(exception)` is called.

If the state machine reaches an `await expr` expression, `expr.GetAwaiter()` is invoked. If the awaiter implements `ICriticalNotifyCompletion` and `IsCompleted` is false, the state machine invokes `builder.AwaitUnsafeOnCompleted(ref awaiter, ref stateMachine)`. `AwaitUnsafeOnCompleted()` should call `awaiter.UnsafeOnCompleted(action)` with an `Action` that calls `stateMachine.MoveNext()` when the awaiter completes. Similarly for `INotifyCompletion` and `builder.AwaitOnCompleted()`.

`SetStateMachine(IAsyncStateMachine)` is called by the compiler-generated `IAsyncStateMachine` implementation. That can be used to identify the instance of the builder associated with a state machine instance, particularly for cases where the state machine is implemented as a value type: if the builder calls `stateMachine.SetStateMachine(stateMachine)`, the `stateMachine` will call `builder.SetStateMachine(stateMachine)` on the *builder instance associated with* `stateMachine`.

Overload Resolution

Overload resolution is extended to recognize *task types* in addition to `Task` and `Task<T>`.

An `async` lambda with no return value is an exact match for an overload candidate parameter of non-generic *task type*, and an `async` lambda with return type `T` is an exact match for an overload candidate parameter of

generic *task type*.

Otherwise if an `async` lambda is not an exact match for either of two candidate parameters of *task types*, or an exact match for both, and there is an implicit conversion from one candidate type to the other, the from candidate wins. Otherwise recursively evaluate the types `A` and `B` within `Task1<A>` and `Task2` for better match.

Otherwise if an `async` lambda is not an exact match for either of two candidate parameters of *task types*, but one candidate is a more specialized type than the other, the more specialized candidate wins.

Async Main

12/28/2021 • 2 minutes to read • [Edit Online](#)

Summary

Allow `await` to be used in an application's Main / entrypoint method by allowing the entrypoint to return `Task` / `Task<int>` and be marked `async`.

Motivation

It is very common when learning C#, when writing console-based utilities, and when writing small test apps to want to call and `await` `async` methods from Main. Today we add a level of complexity here by forcing such `await` 'ing to be done in a separate async method, which causes developers to need to write boilerplate like the following just to get started:

```
public static void Main()
{
    MainAsync().GetAwaiter().GetResult();
}

private static async Task MainAsync()
{
    ... // Main body here
}
```

We can remove the need for this boilerplate and make it easier to get started simply by allowing Main itself to be `async` such that `await` s can be used in it.

Detailed design

The following signatures are currently allowed entrypoints:

```
static void Main()
static void Main(string[])
static int Main()
static int Main(string[])
```

We extend the list of allowed entrypoints to include:

```
static Task Main()
static Task<int> Main()
static Task Main(string[])
static Task<int> Main(string[])
```

To avoid compatibility risks, these new signatures will only be considered as valid entrypoints if no overloads of the previous set are present. The language / compiler will not require that the entrypoint be marked as `async`, though we expect the vast majority of uses will be marked as such.

When one of these is identified as the entrypoint, the compiler will synthesize an actual entrypoint method that calls one of these coded methods:

- `static Task Main()` will result in the compiler emitting the equivalent of

```
private static void $GeneratedMain() => Main().GetAwaiter().GetResult();
```

- `static Task Main(string[])` will result in the compiler emitting the equivalent of

```
private static void $GeneratedMain(string[] args) => Main(args).GetAwaiter().GetResult();
```
- `static Task<int> Main()` will result in the compiler emitting the equivalent of

```
private static int $GeneratedMain() => Main().GetAwaiter().GetResult();
```
- `static Task<int> Main(string[])` will result in the compiler emitting the equivalent of

```
private static int $GeneratedMain(string[] args) => Main(args).GetAwaiter().GetResult();
```

Example usage:

```
using System;
using System.Net.Http;

class Test
{
    static async Task Main(string[] args) =>
        Console.WriteLine(await new HttpClient().GetStringAsync(args[0]));
}
```

Drawbacks

The main drawback is simply the additional complexity of supporting additional entrypoint signatures.

Alternatives

Other variants considered:

Allowing `async void`. We need to keep the semantics the same for code calling it directly, which would then make it difficult for a generated entrypoint to call it (no Task returned). We could solve this by generating two other methods, e.g.

```
public static async void Main()
{
    ... // await code
}
```

becomes

```
public static async void Main() => await $MainTask();

private static void $EntrypointMain() => Main().GetAwaiter().GetResult();

private static async Task $MainTask()
{
    ... // await code
}
```

There are also concerns around encouraging usage of `async void`.

Using "MainAsync" instead of "Main" as the name. While the async suffix is recommended for Task-returning methods, that's primarily about library functionality, which Main is not, and supporting additional entrypoint names beyond "Main" is not worth it.

Unresolved questions

n/a

Design meetings

n/a

Target-typed "default" literal

12/28/2021 • 2 minutes to read • [Edit Online](#)

Summary

The target-typed `default` feature is a shorter form variation of the `default(T)` operator, which allows the type to be omitted. Its type is inferred by target-typing instead. Aside from that, it behaves like `default(T)`.

Motivation

The main motivation is to avoid typing redundant information.

For instance, when invoking `void Method(ImmutableArray<SomeType> array)`, the *default* literal allows `M(default)` in place of `M(default(ImmutableArray<SomeType>))`.

This is applicable in a number of scenarios, such as:

- declaring locals (`ImmutableArray<SomeType> x = default;`)
- ternary operations (`var x = flag ? default : ImmutableArray<SomeType>.Empty;`)
- returning in methods and lambdas (`return default;`)
- declaring default values for optional parameters (`void Method(ImmutableArray<SomeType> arrayOpt = default)`)
- including default values in array creation expressions (`var x = new[] { default, ImmutableArray.Create(y) };`)

Detailed design

A new expression is introduced, the *default* literal. An expression with this classification can be implicitly converted to any type, by a *default literal conversion*.

The inference of the type for the *default* literal works the same as that for the *null* literal, except that any type is allowed (not just reference types).

This conversion produces the default value of the inferred type.

The *default* literal may have a constant value, depending on the inferred type. So `const int x = default;` is legal, but `const int? y = default;` is not.

The *default* literal can be the operand of equality operators, as long as the other operand has a type. So `default == x` and `x == default` are valid expressions, but `default == default` is illegal.

Drawbacks

A minor drawback is that *default* literal can be used in place of *null* literal in most contexts. Two of the exceptions are `throw null;` and `null == null`, which are allowed for the *null* literal, but not the *default* literal.

Alternatives

There are a couple of alternatives to consider:

- The status quo: The feature is not justified on its own merits and developers continue to use the default operator with an explicit type.
- Extending the null literal: This is the VB approach with `Nothing`. We could allow `int x = null;`.

Unresolved questions

- [x] Should *default* be allowed as the operand of the *is* or *as* operators? Answer: disallow `default is T`, allow `x is default`, allow `default as RefType` (with always-null warning)

Design meetings

- [LDM 3/7/2017](#)
- [LDM 3/28/2017](#)
- [LDM 5/31/2017](#)

Infer tuple names (aka. tuple projection initializers)

12/28/2021 • 2 minutes to read • [Edit Online](#)

Summary

In a number of common cases, this feature allows the tuple element names to be omitted and instead be inferred. For instance, instead of typing `(f1: x.f1, f2: x?.f2)`, the element names "f1" and "f2" can be inferred from `(x.f1, x?.f2)`.

This parallels the behavior of anonymous types, which allow inferring member names during creation. For instance, `new { x.f1, y?.f2 }` declares members "f1" and "f2".

This is particularly handy when using tuples in LINQ:

```
// "c" and "result" have element names "f1" and "f2"
var result = list.Select(c => (c.f1, c.f2)).Where(t => t.f2 == 1);
```

Detailed design

There are two parts to the change:

1. Try to infer a candidate name for each tuple element which does not have an explicit name:
 - Using same rules as name inference for anonymous types.
 - In C#, this allows three cases: `y` (identifier), `x.y` (simple member access) and `x?.y` (conditional access).
 - In VB, this allows for additional cases, such as `x.y()`.
 - Rejecting reserved tuple names (case-sensitive in C#, case-insensitive in VB), as they are either forbidden or already implicit. For instance, such as `ItemN`, `Rest`, and `Tostring`.
 - If any candidate names are duplicates (case-sensitive in C#, case-insensitive in VB) within the entire tuple, we drop those candidates,
2. During conversions (which check and warn about dropping names from tuple literals), inferred names would not produce any warnings. This avoids breaking existing tuple code.

Note that the rule for handling duplicates is different than that for anonymous types. For instance, `new { x.f1, x.f1 }` produces an error, but `(x.f1, x.f1)` would still be allowed (just without any inferred names). This avoids breaking existing tuple code.

For consistency, the same would apply to tuples produced by deconstruction-assignments (in C#):

```
// tuple has element names "f1" and "f2"
var tuple = ((x.f1, x?.f2) = (1, 2));
```

The same would also apply to VB tuples, using the VB-specific rules for inferring name from expression and case-insensitive name comparisons.

When using the C# 7.1 compiler (or later) with language version "7.0", the element names will be inferred (despite the feature not being available), but there will be a use-site error for trying to access them. This will limit additions of new code that would later face the compatibility issue (described below).

Drawbacks

The main drawback is that this introduces a compatibility break from C# 7.0:

```
Action y = () => M();  
var t = (x: x, y);  
t.y(); // this might have previously picked up an extension method called "y", but would now call the  
lambda.
```

The compatibility council found this break acceptable, given that it is limited and the time window since tuples shipped (in C# 7.0) is short.

References

- [LDM April 4th 2017](#)
- [Github discussion](#) (thanks @alrz for bringing this issue up)
- [Tuples design](#)

pattern-matching with generics

12/28/2021 • 2 minutes to read • [Edit Online](#)

Summary

The specification for the [existing C# as operator](#) permits there to be no conversion between the type of the operand and the specified type when either is an open type. However, in C# 7 the `Type identifier` pattern requires there be a conversion between the type of the input and the given type.

We propose to relax this and change `expression is Type identifier`, in addition to being permitted in the conditions when it is permitted in C# 7, to also be permitted when `expression as Type` would be allowed. Specifically, the new cases are cases where the type of the expression or the specified type is an open type.

Motivation

Cases where pattern-matching should "obviously" be permitted currently fail to compile. See, for example, <https://github.com/dotnet/roslyn/issues/16195>.

Detailed design

We change the paragraph in the pattern-matching specification (the proposed addition is shown in bold):

Certain combinations of static type of the left-hand-side and the given type are considered incompatible and result in compile-time error. A value of static type `E` is said to be *pattern compatible* with the type `T` if there exists an identity conversion, an implicit reference conversion, a boxing conversion, an explicit reference conversion, or an unboxing conversion from `E` to `T`, **or if either `E` or `T` is an open type**. It is a compile-time error if an expression of type `E` is not pattern compatible with the type in a type pattern that it is matched with.

Drawbacks

None.

Alternatives

None.

Unresolved questions

None.

Design meetings

LDM considered this question and felt it was a bug-fix level change. We are treating it as a separate language feature because just making the change after the language has been released would introduce a forward incompatibility. Using the proposed change requires that the programmer specify language version 7.1.

Readonly references

12/28/2021 • 24 minutes to read • [Edit Online](#)

Summary

The "readonly references" feature is actually a group of features that leverage the efficiency of passing variables by reference, but without exposing the data to modifications:

- `in` parameters
- `ref readonly` returns
- `readonly` structs
- `ref / in` extension methods
- `ref readonly` locals
- `ref` conditional expressions

Passing arguments as readonly references.

There is an existing proposal that touches this topic <https://github.com/dotnet/roslyn/issues/115> as a special case of readonly parameters without going into many details. Here I just want to acknowledge that the idea by itself is not very new.

Motivation

Prior to this feature C# did not have an efficient way of expressing a desire to pass struct variables into method calls for readonly purposes with no intention of modifying. Regular by-value argument passing implies copying, which adds unnecessary costs. That drives users to use by-ref argument passing and rely on comments/documentation to indicate that the data is not supposed to be mutated by the callee. It is not a good solution for many reasons.

The examples are numerous - vector/matrix math operators in graphics libraries like [XNA](#) are known to have ref operands purely because of performance considerations. There is code in Roslyn compiler itself that uses structs to avoid allocations and then passes them by reference to avoid copying costs.

Solution (`in` parameters)

Similarly to the `out` parameters, `in` parameters are passed as managed references with additional guarantees from the callee.

Unlike `out` parameters which *must* be assigned by the callee before any other use, `in` parameters cannot be assigned by the callee at all.

As a result `in` parameters allow for effectiveness of indirect argument passing without exposing arguments to mutations by the callee.

Declaring `in` parameters

`in` parameters are declared by using `in` keyword as a modifier in the parameter signature.

For all purposes the `in` parameter is treated as a `readonly` variable. Most of the restrictions on the use of `in` parameters inside the method are the same as with `readonly` fields.

Indeed an `in` parameter may represent a `readonly` field. Similarity of restrictions is not a coincidence.

For example fields of an `in` parameter which has a struct type are all recursively classified as `readonly`

variables .

```
static Vector3 Add (in Vector3 v1, in Vector3 v2)
{
    // not OK!!
    v1 = default(Vector3);

    // not OK!!
    v1.X = 0;

    // not OK!!
    foo(ref v1.X);

    // OK
    return new Vector3(v1.X + v2.X, v1.Y + v2.Y, v1.Z + v2.Z);
}
```

- `in` parameters are allowed anywhere where ordinary byval parameters are allowed. This includes indexers, operators (including conversions), delegates, lambdas, local functions.

```
(in int x) => x // lambda expression
TValue this[in TKey index]; // indexer
public static Vector3 operator +(in Vector3 x, in Vector3 y) => ... // operator
```

- `in` is not allowed in combination with `out` or with anything that `out` does not combine with.
- It is not permitted to overload on `ref` / `out` / `in` differences.
- It is permitted to overload on ordinary byval and `in` differences.
- For the purpose of OHI (Overloading, Hiding, Implementing), `in` behaves similarly to an `out` parameter. All the same rules apply. For example the overriding method will have to match `in` parameters with `in` parameters of an identity-convertible type.
- For the purpose of delegate/lambda/method group conversions, `in` behaves similarly to an `out` parameter. Lambdas and applicable method group conversion candidates will have to match `in` parameters of the target delegate with `in` parameters of an identity-convertible type.
- For the purpose of generic variance, `in` parameters are nonvariant.

NOTE: There are no warnings on `in` parameters that have reference or primitives types. It may be pointless in general, but in some cases user must/want to pass primitives as `in`. Examples - overriding a generic method like `Method(in T param)` when `T` was substituted to be `int`, or when having methods like `Volatile.Read(in int location)`

It is conceivable to have an analyzer that warns in cases of inefficient use of `in` parameters, but the rules for such analysis would be too fuzzy to be a part of a language specification.

Use of `in` at call sites. (`in` arguments)

There are two ways to pass arguments to `in` parameters.

`in` arguments can match `in` parameters:

An argument with an `in` modifier at the call site can match `in` parameters.

```

int x = 1;

void M1<T>(in T x)
{
    // . . .
}

var x = M1(in x); // in argument to a method

class D
{
    public string this[in Guid index];
}

D dictionary = . . . ;
var y = dictionary[in Guid.Empty]; // in argument to an indexer

```

- `in` argument must be a *readable* LValue(*). Example: `M1(in 42)` is invalid

(*) The notion of LValue/RValue vary between languages.

Here, by LValue I mean an expression that represent a location that can be referred to directly. And RValue means an expression that yields a temporary result which does not persist on its own.

- In particular it is valid to pass `readonly` fields, `in` parameters or other formally `readonly` variables as `in` arguments. Example: `dictionary[in Guid.Empty]` is legal. `Guid.Empty` is a static readonly field.
- `in` argument must have type *identity-convertible* to the type of the parameter. Example: `M1<object>(in Guid.Empty)` is invalid. `Guid.Empty` is not *identity-convertible* to `object`

The motivation for the above rules is that `in` arguments guarantee *aliasing* of the argument variable. The callee always receives a direct reference to the same location as represented by the argument.

- in rare situations when `in` arguments must be stack-spilled due to `await` expressions used as operands of the same call, the behavior is the same as with `out` and `ref` arguments - if the variable cannot be spilled in referentially-transparent manner, an error is reported.

Examples:

1. `M1(in staticField, await SomethingAsync())` is valid. `staticField` is a static field which can be accessed more than once without observable side effects. Therefore both the order of side effects and aliasing requirements can be provided.
2. `M1(in RefReturningMethod(), await SomethingAsync())` will produce an error. `RefReturningMethod()` is a `ref` returning method. A method call may have observable side effects, therefore it must be evaluated before the `SomethingAsync()` operand. However the result of the invocation is a reference that cannot be preserved across the `await` suspension point which make the direct reference requirement impossible.

NOTE: the stack spilling errors are considered to be implementation-specific limitations. Therefore they do not have effect on overload resolution or lambda inference.

Ordinary byval arguments can match `in` parameters:

Regular arguments without modifiers can match `in` parameters. In such case the arguments have the same relaxed constraints as an ordinary byval arguments would have.

The motivation for this scenario is that `in` parameters in APIs may result in inconveniences for the user when arguments cannot be passed as a direct reference - ex: literals, computed or `await`-ed results or arguments that happen to have more specific types.

All these cases have a trivial solution of storing the argument value in a temporary local of appropriate type and passing that local as an `in` argument.

To reduce the need for such boilerplate code compiler can perform the same transformation, if needed, when `in` modifier is not present at the call site.

In addition, in some cases, such as invocation of operators, or `in` extension methods, there is no syntactical way to specify `in` at all. That alone requires specifying the behavior of ordinary byval arguments when they match `in` parameters.

In particular:

- it is valid to pass RValues. A reference to a temporary is passed in such case. Example:

```
Print("hello");           // not an error.

void Print<T>(in T x)
{
    // . . .
}
```

- implicit conversions are allowed.

This is actually a special case of passing an RValue

A reference to a temporary holding converted value is passed in such case. Example:

```
Print<int>(Short.MaxValue)    // not an error.
```

- in a case of a receiver of an `in` extension method (as opposed to `ref` extension methods), RValues or implicit *this-argument-conversions* are allowed. A reference to a temporary holding converted value is passed in such case. Example:

```
public static IEnumerable<T> Concat<T>(in this (IEnumerable<T>, IEnumerable<T>) arg) => . . .;

("aa", "bb").Concat<char>()    // not an error.
```

More information on `ref`/`in` extension methods is provided further in this document.

- argument spilling due to `await` operands could spill "by-value", if necessary. In scenarios where providing a direct reference to the argument is not possible due to intervening `await` a copy of the argument's value is spilled instead.

Example:

```
M1(RefReturningMethod(), await SomethingAsync())    // not an error.
```

Since the result of a side-effecting invocation is a reference that cannot be preserved across `await` suspension, a temporary containing the actual value will be preserved instead (as it would in an ordinary byval parameter case).

Omitted optional arguments

It is permitted for an `in` parameter to specify a default value. That makes the corresponding argument optional.

Omitting optional argument at the call site results in passing the default value via a temporary.

```
Print("hello");          // not an error, same as
Print("hello", c: Color.Black);

void Print(string s, in Color c = Color.Black)
{
    // . . .
}
```

Aliasing behavior in general

Just like `ref` and `out` variables, `in` variables are references/aliases to existing locations.

While callee is not allowed to write into them, reading an `in` parameter can observe different values as a side effect of other evaluations.

Example:

```
static Vector3 v = Vector3.Unity;

static void Main()
{
    Test(v);
}

static void Test(in Vector3 v1)
{
    Debug.Assert(v1 == Vector3.Unity);
    // changes v1 deterministically (no races required)
    ChangeV();
    Debug.Assert(v1 == Vector3.UnityX);
}

static void ChangeV()
{
    v = Vector3.UnityX;
}
```

`in` parameters and capturing of local variables.

For the purpose of lambda/async capturing `in` parameters behave the same as `out` and `ref` parameters.

- `in` parameters cannot be captured in a closure
- `in` parameters are not allowed in iterator methods
- `in` parameters are not allowed in async methods

Temporary variables.

Some uses of `in` parameter passing may require indirect use of a temporary local variable:

- `in` arguments are always passed as direct aliases when call-site uses `in`. Temporary is never used in such case.
- `in` arguments are not required to be direct aliases when call-site does not use `in`. When argument is not an LValue, a temporary may be used.
- `in` parameter may have default value. When corresponding argument is omitted at the call site, the default value are passed via a temporary.
- `in` arguments may have implicit conversions, including those that do not preserve identity. A temporary is used in those cases.
- receivers of ordinary struct calls may not be writeable LValues (**existing case!**). A temporary is used in those cases.

The life time of the argument temporaries matches the closest encompassing scope of the call-site.

The formal life time of temporary variables is semantically significant in scenarios involving escape analysis of variables returned by reference.

Metadata representation of `in` parameters.

When `System.Runtime.CompilerServices.IsReadOnlyAttribute` is applied to a byref parameter, it means that the parameter is an `in` parameter.

In addition, if the method is *abstract* or *virtual*, then the signature of such parameters (and only such parameters) must have `modreq[System.Runtime.InteropServices.InAttribute]`.

Motivation: this is done to ensure that in a case of method overriding/implementing the `in` parameters match.

Same requirements apply to `Invoke` methods in delegates.

Motivation: this is to ensure that existing compilers cannot simply ignore `readonly` when creating or assigning delegates.

Returning by readonly reference.

Motivation

The motivation for this sub-feature is roughly symmetrical to the reasons for the `in` parameters - avoiding copying, but on the returning side. Prior to this feature, a method or an indexer had two options: 1) return by reference and be exposed to possible mutations or 2) return by value which results in copying.

Solution (`ref readonly` returns)

The feature allows a member to return variables by reference without exposing them to mutations.

Declaring `ref readonly` returning members

A combination of modifiers `ref readonly` on the return signature is used to indicate that the member returns a readonly reference.

For all purposes a `ref readonly` member is treated as a `readonly` variable - similar to `readonly` fields and `in` parameters.

For example fields of `ref readonly` member which has a struct type are all recursively classified as `readonly` variables. - It is permitted to pass them as `in` arguments, but not as `ref` or `out` arguments.

```
ref readonly Guid Method1()
{
}

Method2(in Method1()); // valid. Can pass as `in` argument.

Method3(ref Method1()); // not valid. Cannot pass as `ref` argument
```

- `ref readonly` returns are allowed in the same places where `ref` returns are allowed. This includes indexers, delegates, lambdas, local functions.
- It is not permitted to overload on `ref` / `ref readonly` / differences.
- It is permitted to overload on ordinary byval and `ref readonly` return differences.
- For the purpose of OHI (Overloading, Hiding, Implementing), `ref readonly` is similar but distinct from `ref`. For example the a method that overrides `ref readonly` one, must itself be `ref readonly` and have

identity-convertible type.

- For the purpose of delegate/lambda/method group conversions, `ref readonly` is similar but distinct from `ref`. Lambdas and applicable method group conversion candidates have to match `ref readonly` return of the target delegate with `ref readonly` return of the type that is identity-convertible.
- For the purpose of generic variance, `ref readonly` returns are nonvariant.

NOTE: There are no warnings on `ref readonly` returns that have reference or primitives types. It may be pointless in general, but in some cases user must/want to pass primitives as `in`. Examples - overriding a generic method like `ref readonly T Method()` when `T` was substituted to be `int`.

It is conceivable to have an analyzer that warns in cases of inefficient use of `ref readonly` returns, but the rules for such analysis would be too fuzzy to be a part of a language specification.

Returning from `ref readonly` members

Inside the method body the syntax is the same as with regular `ref` returns. The `readonly` will be inferred from the containing method.

The motivation is that `return ref readonly <expression>` is unnecessary long and only allows for mismatches on the `readonly` part that would always result in errors. The `ref` is, however, required for consistency with other scenarios where something is passed via strict aliasing vs. by value.

Unlike the case with `in` parameters, `ref readonly` returns never return via a local copy. Considering that the copy would cease to exist immediately upon returning such practice would be pointless and dangerous. Therefore `ref readonly` returns are always direct references.

Example:

```
struct ImmutableArray<T>
{
    private readonly T[] array;

    public ref readonly T ItemRef(int i)
    {
        // returning a readonly reference to an array element
        return ref this.array[i];
    }
}
```

- An argument of `return ref` must be an LValue (**existing rule**)
- An argument of `return ref` must be "safe to return" (**existing rule**)
- In a `ref readonly` member an argument of `return ref` is *not required to be writeable*. For example such member can `ref`-return a `readonly` field or one of its `in` parameters.

Safe to Return rules.

Normal safe to return rules for references will apply to `readonly` references as well.

Note that a `ref readonly` can be obtained from a regular `ref` local/parameter/return, but not the other way around. Otherwise the safety of `ref readonly` returns is inferred the same way as for regular `ref` returns.

Considering that RValues can be passed as `in` parameter and returned as `ref readonly` we need one more rule - **RValues are not safe-to-return by reference.**

Consider the situation when an RValue is passed to an `in` parameter via a copy and then returned back in a

form of a `ref readonly`. In the context of the caller the result of such invocation is a reference to local data and as such is unsafe to return. Once RValues are not safe to return, the existing rule `#6` already handles this case.

Example:

```
ref readonly Vector3 Test1()
{
    // can pass an RValue as "in" (via a temp copy)
    // but the result is not safe to return
    // because the RValue argument was not safe to return by reference
    return ref Test2(default(Vector3));
}

ref readonly Vector3 Test2(in Vector3 r)
{
    // this is ok, r is returnable
    return ref r;
}
```

Updated `safe to return` rules:

1. **refs to variables on the heap are safe to return**
2. **ref/in parameters are safe to return** `in` parameters naturally can only be returned as readonly.
3. **out parameters are safe to return** (but must be definitely assigned, as is already the case today)
4. **instance struct fields are safe to return as long as the receiver is safe to return**
5. **'this' is not safe to return from struct members**
6. **a ref, returned from another method is safe to return if all refs/outs passed to that method as formal parameters were safe to return.** *Specifically it is irrelevant if receiver is safe to return, regardless whether receiver is a struct, class or typed as a generic type parameter.*
7. **RValues are not safe to return by reference.** *Specifically RValues are safe to pass as in parameters.*

NOTE: There are additional rules regarding safety of returns that come into play when ref-like types and ref-reassignments are involved. The rules equally apply to `ref` and `ref readonly` members and therefore are not mentioned here.

Aliasing behavior.

`ref readonly` members provide the same aliasing behavior as ordinary `ref` members (except for being readonly). Therefore for the purpose of capturing in lambdas, async, iterators, stack spilling etc... the same restrictions apply. - I.E. due to inability to capture the actual references and due to side-effecting nature of member evaluation such scenarios are disallowed.

It is permitted and required to make a copy when `ref readonly` return is a receiver of regular struct methods, which take `this` as an ordinary writeable reference. Historically in all cases where such invocations are applied to readonly variable a local copy is made.

Metadata representation.

When `System.Runtime.CompilerServices.IsReadOnlyAttribute` is applied to the return of a byref returning method, it means that the method returns a readonly reference.

In addition, the result signature of such methods (and only those methods) must have

```
modreq[System.Runtime.CompilerServices.IsReadOnlyAttribute].
```

Motivation: this is to ensure that existing compilers cannot simply ignore `readonly` when invoking methods

with `ref readonly` returns

Readonly structs

In short - a feature that makes `this` parameter of all instance members of a struct, except for constructors, an `in` parameter.

Motivation

Compiler must assume that any method call on a struct instance may modify the instance. Indeed a writeable reference is passed to the method as `this` parameter and fully enables this behavior. To allow such invocations on `readonly` variables, the invocations are applied to temp copies. That could be unintuitive and sometimes forces people to abandon `readonly` for performance reasons.

Example: <https://codeblog.jonskeet.uk/2014/07/16/micro-optimization-the-surprising-inefficiency-of-readonly-fields/>

After adding support for `in` parameters and `ref readonly` returns the problem of defensive copying will get worse since readonly variables will become more common.

Solution

Allow `readonly` modifier on struct declarations which would result in `this` being treated as `in` parameter on all struct instance methods except for constructors.

```
static void Test(in Vector3 v1)
{
    // no need to make a copy of v1 since Vector3 is a readonly struct
    System.Console.WriteLine(v1.ToString());
}

readonly struct Vector3
{
    . . .

    public override string ToString()
    {
        // not OK!! `this` is an `in` parameter
        foo(ref this.X);

        // OK
        return $"X: {X}, Y: {Y}, Z: {Z}";
    }
}
```

Restrictions on members of readonly struct

- Instance fields of a readonly struct must be readonly.
Motivation: can only be written to externally, but not through members.
- Instance autoproperties of a readonly struct must be get-only.
Motivation: consequence of restriction on instance fields.
- Readonly struct may not declare field-like events.
Motivation: consequence of restriction on instance fields.

Metadata representation.

When `System.Runtime.CompilerServices.IsReadOnlyAttribute` is applied to a value type, it means that the type is a `readonly struct`.

In particular:

- The identity of the `IsReadOnlyAttribute` type is unimportant. In fact it can be embedded by the compiler in

the containing assembly if needed.

ref / in extension methods

There is actually an existing proposal (<https://github.com/dotnet/roslyn/issues/165>) and corresponding prototype PR (<https://github.com/dotnet/roslyn/pull/15650>). I just want to acknowledge that this idea is not entirely new. It is, however, relevant here since `ref readonly` elegantly removes the most contentious issue about such methods - what to do with RValue receivers.

The general idea is allowing extension methods to take the `this` parameter by reference, as long as the type is known to be a struct type.

```
public static void Extension(ref this Guid self)
{
    // do something
}
```

The reasons for writing such extension methods are primarily:

1. Avoid copying when receiver is a large struct
2. Allow mutating extension methods on structs

The reasons why we do not want to allow this on classes

1. It would be of very limited purpose.
2. It would break long standing invariant that a method call cannot turn non-`null` receiver to become `null` after invocation.

In fact, currently a non-`null` variable cannot become `null` unless *explicitly* assigned or passed by `ref` or `out`. That greatly aids readability or other forms of "can this be a null here" analysis. 3. It would be hard to reconcile with "evaluate once" semantics of null-conditional accesses. Example:

`obj.stringField?.RefExtension(...)` - need to capture a copy of `stringField` to make the null check meaningful, but then assignments to `this` inside `RefExtension` would not be reflected back to the field.

An ability to declare extension methods on **structs** that take the first argument by reference was a long-standing request. One of the blocking consideration was "what happens if receiver is not an LValue?".

- There is a precedent that any extension method could also be called as a static method (sometimes it is the only way to resolve ambiguity). It would dictate that RValue receivers should be disallowed.
- On the other hand there is a practice of making invocation on a copy in similar situations when struct instance methods are involved.

The reason why the "implicit copying" exists is because the majority of struct methods do not actually modify the struct while not being able to indicate that. Therefore the most practical solution was to just make the invocation on a copy, but this practice is known for harming performance and causing bugs.

Now, with availability of `in` parameters, it is possible for an extension to signal the intent. Therefore the conundrum can be resolved by requiring `ref` extensions to be called with writeable receivers while `in` extensions permit implicit copying if necessary.

```
// this can be called on either RValue or an LValue
public static void Reader(in this Guid self)
{
    // do something nonmutating.
    WriteLine(self == default(Guid));
}

// this can be called only on an LValue
public static void Mutator(ref this Guid self)
{
    // can mutate self
    self = new Guid();
}
```

in **extensions and generics.**

The purpose of `ref` extension methods is to mutate the receiver directly or by invoking mutating members. Therefore `ref this T` extensions are allowed as long as `T` is constrained to be a struct.

On the other hand `in` extension methods exist specifically to reduce implicit copying. However any use of an `in T` parameter will have to be done through an interface member. Since all interface members are considered mutating, any such use would require a copy. - Instead of reducing copying, the effect would be the opposite. Therefore `in this T` is not allowed when `T` is a generic type parameter regardless of constraints.

Valid kinds of extension methods (recap):

The following forms of `this` declaration in an extension method are now allowed:

1. `this T arg` - regular byval extension. (existing case)
 - T can be any type, including reference types or type parameters. Instance will be the same variable after the call. Allows implicit conversions of *this-argument-conversion* kind. Can be called on RValues.
 - `in this T self` - `in` extension. T must be an actual struct type. Instance will be the same variable after the call. Allows implicit conversions of *this-argument-conversion* kind. Can be called on RValues (may be invoked on a temp if needed).
 - `ref this T self` - `ref` extension. T must be a struct type or a generic type parameter constrained to be a struct. Instance may be written to by the invocation. Allows only identity conversions. Must be called on writeable LValue. (never invoked via a temp).

Readonly ref locals.

Motivation.

Once `ref readonly` members were introduced, it was clear from the use that they need to be paired with appropriate kind of local. Evaluation of a member may produce or observe side effects, therefore if the result must be used more than once, it needs to be stored. Ordinary `ref` locals do not help here since they cannot be assigned a `readonly` reference.

Solution.

Allow declaring `ref readonly` locals. This is a new kind of `ref` locals that is not writeable. As a result `ref readonly` locals can accept references to readonly variables without exposing these variables to writes.

Declaring and using `ref readonly` locals.

The syntax of such locals uses `ref readonly` modifiers at declaration site (in that specific order). Similarly to ordinary `ref` locals, `ref readonly` locals must be ref-initialized at declaration. Unlike regular `ref` locals, `ref readonly` locals can refer to `readonly` LValues like `in` parameters, `readonly` fields, `ref readonly`

methods.

For all purposes a `ref readonly` local is treated as a `readonly` variable. Most of the restrictions on the use are the same as with `readonly` fields or `in` parameters.

For example fields of an `in` parameter which has a struct type are all recursively classified as `readonly` variables .

```
static readonly ref Vector3 M1() => . . .

static readonly ref Vector3 M1_Trace()
{
    // OK
    ref readonly var r1 = ref M1();

    // Not valid. Need an LValue
    ref readonly Vector3 r2 = ref default(Vector3);

    // Not valid. r1 is readonly.
    Mutate(ref r1);

    // OK.
    Print(in r1);

    // OK.
    return ref r1;
}
```

Restrictions on use of `ref readonly` locals

Except for their `readonly` nature, `ref readonly` locals behave like ordinary `ref` locals and are subject to exactly same restrictions.

For example restrictions related to capturing in closures, declaring in `async` methods or the `safe-to-return` analysis equally applies to `ref readonly` locals.

Ternary `ref` expressions. (aka "Conditional LValues")

Motivation

Use of `ref` and `ref readonly` locals exposed a need to ref-initialize such locals with one or another target variable based on a condition.

A typical workaround is to introduce a method like:

```
ref T Choice(bool condition, ref T consequence, ref T alternative)
{
    if (condition)
    {
        return ref consequence;
    }
    else
    {
        return ref alternative;
    }
}
```

Note that `Choice` is not an exact replacement of a ternary since *all* arguments must be evaluated at the call site, which was leading to unintuitive behavior and bugs.

The following will not work as expected:

```
// will crash with NRE because 'arr[0]' will be executed unconditionally
ref var r = ref Choice(arr != null, ref arr[0], ref otherArr[0]);
```

Solution

Allow special kind of conditional expression that evaluates to a reference to one of LValue argument based on a condition.

Using `ref` **ternary expression**.

The syntax for the `ref` flavor of a conditional expression is

```
<condition> ? ref <consequence> : ref <alternative>;
```

Just like with the ordinary conditional expression only `<consequence>` or `<alternative>` is evaluated depending on result of the boolean condition expression.

Unlike ordinary conditional expression, `ref` conditional expression:

- requires that `<consequence>` and `<alternative>` are LValues.
- `ref` conditional expression itself is an LValue and
- `ref` conditional expression is writeable if both `<consequence>` and `<alternative>` are writeable LValues

Examples:

`ref` ternary is an LValue and as such it can be passed/assigned/returned by reference;

```
// pass by reference
foo(ref (arr != null ? ref arr[0]: ref otherArr[0]));

// return by reference
return ref (arr != null ? ref arr[0]: ref otherArr[0]);
```

Being an LValue, it can also be assigned to.

```
// assign to
(arr != null ? ref arr[0]: ref otherArr[0]) = 1;

// error. readOnlyField is readonly and thus conditional expression is readonly
(arr != null ? ref arr[0]: ref obj.readOnlyField) = 1;
```

Can be used as a receiver of a method call and skip copying if necessary.

```
// no copies
(arr != null ? ref arr[0]: ref otherArr[0]).StructMethod();

// invoked on a copy.
// The receiver is `readonly` because readOnlyField is readonly.
(arr != null ? ref arr[0]: ref obj.readOnlyField).StructMethod();

// no copies. `ReadOnlyStructMethod` is a method on a `readonly` struct
// and can be invoked directly on a readonly receiver
(arr != null ? ref arr[0]: ref obj.readOnlyField).ReadOnlyStructMethod();
```

`ref` ternary can be used in a regular (not ref) context as well.

```
// only an example
// a regular ternary could work here just the same
int x = (arr != null ? ref arr[0]: ref otherArr[0]);
```

Drawbacks

I can see two major arguments against enhanced support for references and readonly references:

1. The problems that are solved here are very old. Why suddenly solve them now, especially since it would not help existing code?

As we find C# and .Net used in new domains, some problems become more prominent.

As examples of environments that are more critical than average about computation overheads, I can list

- cloud/datacenter scenarios where computation is billed for and responsiveness is a competitive advantage.
- Games/VR/AR with soft-realtime requirements on latencies

This feature does not sacrifice any of the existing strengths such as type-safety, while allowing to lower overheads in some common scenarios.

2. Can we reasonably guarantee that the callee will play by the rules when it opts into `readonly` contracts?

We have similar trust when using `out`. Incorrect implementation of `out` can cause unspecified behavior, but in reality it rarely happens.

Making the formal verification rules familiar with `ref readonly` would further mitigate the trust issue.

Alternatives

The main competing design is really "do nothing".

Unresolved questions

Design meetings

<https://github.com/dotnet/csharplang/blob/master/meetings/2017/LDM-2017-02-22.md>

<https://github.com/dotnet/csharplang/blob/master/meetings/2017/LDM-2017-03-01.md>

<https://github.com/dotnet/csharplang/blob/master/meetings/2017/LDM-2017-08-28.md>

<https://github.com/dotnet/csharplang/blob/master/meetings/2017/LDM-2017-09-25.md>

<https://github.com/dotnet/csharplang/blob/master/meetings/2017/LDM-2017-09-27.md>

Compile time enforcement of safety for ref-like types

12/28/2021 • 16 minutes to read • [Edit Online](#)

Introduction

The main reason for the additional safety rules when dealing with types like `Span<T>` and `ReadOnlySpan<T>` is that such types must be confined to the execution stack.

There are two reasons why `Span<T>` and similar types must be a stack-only types.

1. `Span<T>` is semantically a struct containing a reference and a range - `(ref T data, int length)`. Regardless of actual implementation, writes to such struct would not be atomic. Concurrent "tearing" of such struct would lead to the possibility of `length` not matching the `data`, causing out-of-range accesses and type-safety violations, which ultimately could result in GC heap corruption in seemingly "safe" code.
2. Some implementations of `Span<T>` literally contain a managed pointer in one of its fields. Managed pointers are not supported as fields of heap objects and code that manages to put a managed pointer on the GC heap typically crashes at JIT time.

All the above problems would be alleviated if instances of `Span<T>` are constrained to exist only on the execution stack.

An additional problem arises due to composition. It would be generally desirable to build more complex data types that would embed `Span<T>` and `ReadOnlySpan<T>` instances. Such composite types would have to be structs and would share all the hazards and requirements of `Span<T>`. As a result the safety rules described here should be viewed as applicable to the whole range of *ref-like types*.

The [draft language specification](#) is intended to ensure that values of a ref-like type occur only on the stack.

Generalized `ref-like` types in source code

`ref-like` structs are explicitly marked in the source code using `ref` modifier:

```
ref struct TwoSpans<T>
{
    // can have ref-like instance fields
    public Span<T> first;
    public Span<T> second;
}

// error: arrays of ref-like types are not allowed.
TwoSpans<T>[] arr = null;
```

Designating a struct as ref-like will allow the struct to have ref-like instance fields and will also make all the requirements of ref-like types applicable to the struct.

Metadata representation of ref-like structs

Ref-like structs will be marked with `System.Runtime.CompilerServices.IsRefLikeAttribute` attribute.

The attribute will be added to common base libraries such as `mscorlib`. In a case if the attribute is not available,

compiler will generate an internal one similarly to other embedded-on-demand attributes such as

`IsReadOnlyAttribute`.

An additional measure will be taken to prevent the use of ref-like structs in compilers not familiar with the safety rules (this includes C# compilers prior to the one in which this feature is implemented).

Having no other good alternatives that work in old compilers without servicing, an `Obsolete` attribute with a known string will be added to all ref-like structs. Compilers that know how to use ref-like types will ignore this particular form of `Obsolete`.

A typical metadata representation:

```
[IsRefLike]
[Obsolete("Types with embedded references are not supported in this version of your compiler.")]
public struct TwoSpans<T>
{
    // . . . .
}
```

NOTE: it is not the goal to make it so that any use of ref-like types on old compilers fails 100%. That is hard to achieve and is not strictly necessary. For example there would always be a way to get around the `Obsolete` using dynamic code or, for example, creating an array of ref-like types through reflection.

In particular, if user wants to actually put an `Obsolete` or `Deprecated` attribute on a ref-like type, we will have no choice other than not emitting the predefined one since `Obsolete` attribute cannot be applied more than once..

Examples:

```

SpanLikeType M1(ref SpanLikeType x, Span<byte> y)
{
    // this is all valid, unconcerned with stack-referring stuff
    var local = new SpanLikeType(y);
    x = local;
    return x;
}

void Test1(ref SpanLikeType param1, Span<byte> param2)
{
    Span<byte> stackReferring1 = stackalloc byte[10];
    var stackReferring2 = new SpanLikeType(stackReferring1);

    // this is allowed
    stackReferring2 = M1(ref stackReferring2, stackReferring1);

    // this is NOT allowed
    stackReferring2 = M1(ref param1, stackReferring1);

    // this is NOT allowed
    param1 = M1(ref stackReferring2, stackReferring1);

    // this is NOT allowed
    param2 = stackReferring1.Slice(10);

    // this is allowed
    param1 = new SpanLikeType(param2);

    // this is allowed
    stackReferring2 = param1;
}

ref SpanLikeType M2(ref SpanLikeType x)
{
    return ref x;
}

ref SpanLikeType Test2(ref SpanLikeType param1, Span<byte> param2)
{
    Span<byte> stackReferring1 = stackalloc byte[10];
    var stackReferring2 = new SpanLikeType(stackReferring1);

    ref var stackReferring3 = M2(ref stackReferring2);

    // this is allowed
    stackReferring3 = M1(ref stackReferring2, stackReferring1);

    // this is allowed
    M2(ref stackReferring3) = stackReferring2;

    // this is NOT allowed
    M1(ref param1) = stackReferring2;

    // this is NOT allowed
    param1 = stackReferring3;

    // this is NOT allowed
    return ref stackReferring3;

    // this is allowed
    return ref param1;
}

```

Draft language specification

Below we describe a set of safety rules for ref-like types (`ref struct`s) to ensure that values of these types occur only on the stack. A different, simpler set of safety rules would be possible if locals cannot be passed by reference. This specification would also permit the safe reassignment of ref locals.

Overview

We associate with each expression at compile-time the concept of what scope that expression is permitted to escape to, "safe-to-escape". Similarly, for each lvalue we maintain a concept of what scope a reference to it is permitted to escape to, "ref-safe-to-escape". For a given lvalue expression, these may be different.

These are analogous to the "safe to return" of the ref locals feature, but it is more fine-grained. Where the "safe-to-return" of an expression records only whether (or not) it may escape the enclosing method as a whole, the safe-to-escape records which scope it may escape to (which scope it may not escape beyond). The basic safety mechanism is enforced as follows. Given an assignment from an expression E1 with a safe-to-escape scope S1, to an (lvalue) expression E2 with safe-to-escape scope S2, it is an error if S2 is a wider scope than S1. By construction, the two scopes S1 and S2 are in a nesting relationship, because a legal expression is always safe-to-return from some scope enclosing the expression.

For the time being it is sufficient, for the purpose of the analysis, to support just two scopes - external to the method, and top-level scope of the method. That is because ref-like values with inner scopes cannot be created and ref locals do not support re-assignment. The rules, however, can support more than two scope levels.

The precise rules for computing the *safe-to-return* status of an expression, and the rules governing the legality of expressions, follow.

ref-safe-to-escape

The *ref-safe-to-escape* is a scope, enclosing an lvalue expression, to which it is safe for a ref to the lvalue to escape to. If that scope is the entire method, we say that a ref to the lvalue is *safe to return* from the method.

safe-to-escape

The *safe-to-escape* is a scope, enclosing an expression, to which it is safe for the value to escape to. If that scope is the entire method, we say that the value is *safe to return* from the method.

An expression whose type is not a `ref struct` type is *safe-to-return* from the entire enclosing method. Otherwise we refer to the rules below.

Parameters

An lvalue designating a formal parameter is *ref-safe-to-escape* (by reference) as follows:

- If the parameter is a `ref`, `out`, or `in` parameter, it is *ref-safe-to-escape* from the entire method (e.g. by a `return ref` statement); otherwise
- If the parameter is the `this` parameter of a struct type, it is *ref-safe-to-escape* to the top-level scope of the method (but not from the entire method itself); [Sample](#)
- Otherwise the parameter is a value parameter, and it is *ref-safe-to-escape* to the top-level scope of the method (but not from the method itself).

An expression that is an rvalue designating the use of a formal parameter is *safe-to-escape* (by value) from the entire method (e.g. by a `return` statement). This applies to the `this` parameter as well.

Locals

An lvalue designating a local variable is *ref-safe-to-escape* (by reference) as follows:

- If the variable is a `ref` variable, then its *ref-safe-to-escape* is taken from the *ref-safe-to-escape* of its initializing expression; otherwise
- The variable is *ref-safe-to-escape* the scope in which it was declared.

An expression that is an rvalue designating the use of a local variable is *safe-to-escape* (by value) as follows:

- But the general rule above, a local whose type is not a `ref struct` type is *safe-to-return* from the entire enclosing method.
- If the variable is an iteration variable of a `foreach` loop, then the variable's *safe-to-escape* scope is the same as the *safe-to-escape* of the `foreach` loop's expression.
- A local of `ref struct` type and uninitialized at the point of declaration is *safe-to-return* from the entire enclosing method.
- Otherwise the variable's type is a `ref struct` type, and the variable's declaration requires an initializer. The variable's *safe-to-escape* scope is the same as the *safe-to-escape* of its initializer.

Field reference

An lvalue designating a reference to a field, `e.F`, is *ref-safe-to-escape* (by reference) as follows:

- If `e` is of a reference type, it is *ref-safe-to-escape* from the entire method; otherwise
- If `e` is of a value type, its *ref-safe-to-escape* is taken from the *ref-safe-to-escape* of `e`.

An rvalue designating a reference to a field, `e.F`, has a *safe-to-escape* scope that is the same as the *safe-to-escape* of `e`.

Operators including `?:`

The application of a user-defined operator is treated as a method invocation.

For an operator that yields an rvalue, such as `e1 + e2` or `c ? e1 : e2`, the *safe-to-escape* of the result is the narrowest scope among the *safe-to-escape* of the operands of the operator. As a consequence, for a unary operator that yields an rvalue, such as `+e`, the *safe-to-escape* of the result is the *safe-to-escape* of the operand.

For an operator that yields an lvalue, such as `c ? ref e1 : ref e2`

- the *ref-safe-to-escape* of the result is the narrowest scope among the *ref-safe-to-escape* of the operands of the operator.
- the *safe-to-escape* of the operands must agree, and that is the *safe-to-escape* of the resulting lvalue.

Method invocation

An lvalue resulting from a ref-returning method invocation `e1.M(e2, ...)` is *ref-safe-to-escape* the smallest of the following scopes:

- The entire enclosing method
- the *ref-safe-to-escape* of all `ref` and `out` argument expressions (excluding the receiver)
- For each `in` parameter of the method, if there is a corresponding expression that is an lvalue, its *ref-safe-to-escape*, otherwise the nearest enclosing scope
- the *safe-to-escape* of all argument expressions (including the receiver)

Note: the last bullet is necessary to handle code such as

```
var sp = new Span(...)  
return ref sp[0];
```

or

```
return ref M(sp, 0);
```

An rvalue resulting from a method invocation `e1.M(e2, ...)` is *safe-to-escape* from the smallest of the following scopes:

- The entire enclosing method
- the *safe-to-escape* of all argument expressions (including the receiver)

An Rvalue

An rvalue is *ref-safe-to-escape* from the nearest enclosing scope. This occurs for example in an invocation such as `M(ref d.Length)` where `d` is of type `dynamic`. It is also consistent with (and perhaps subsumes) our handling of arguments corresponding to `in` parameters.

Property invocations

A property invocation (either `get` or `set`) is treated as a method invocation of the underlying method by the above rules.

`stackalloc`

A `stackalloc` expression is an rvalue that is *safe-to-escape* to the top-level scope of the method (but not from the entire method itself).

Constructor invocations

A `new` expression that invokes a constructor obeys the same rules as a method invocation that is considered to return the type being constructed.

In addition *safe-to-escape* is no wider than the smallest of the *safe-to-escape* of all arguments/operands of the object initializer expressions, recursively, if initializer is present.

Span constructor

The language relies on `Span<T>` not having a constructor of the following form:

```
void Example(ref int x)
{
    // Create a span of length one
    var span = new Span<int>(ref x);
}
```

Such a constructor makes `Span<T>` which are used as fields indistinguishable from a `ref` field. The safety rules described in this document depend on `ref` fields not being a valid construct in C# or .NET.

`default` **expressions**

A `default` expression is *safe-to-escape* from the entire enclosing method.

Language Constraints

We wish to ensure that no `ref` local variable, and no variable of `ref struct` type, refers to stack memory or variables that are no longer alive. We therefore have the following language constraints:

- Neither a `ref` parameter, nor a `ref` local, nor a parameter or local of a `ref struct` type can be lifted into a lambda or local function.
- Neither a `ref` parameter nor a parameter of a `ref struct` type may be an argument on an iterator method or an `async` method.
- Neither a `ref` local, nor a local of a `ref struct` type may be in scope at the point of a `yield return` statement or an `await` expression.
- A `ref struct` type may not be used as a type argument, or as an element type in a tuple type.
- A `ref struct` type may not be the declared type of a field, except that it may be the declared type of an instance field of another `ref struct`.

- A `ref struct` type may not be the element type of an array.
- A value of a `ref struct` type may not be boxed:
 - There is no conversion from a `ref struct` type to the type `object` or the type `System.ValueType`.
 - A `ref struct` type may not be declared to implement any interface
 - No instance method declared in `object` or in `System.ValueType` but not overridden in a `ref struct` type may be called with a receiver of that `ref struct` type.
 - No instance method of a `ref struct` type may be captured by method conversion to a delegate type.
- For a ref reassignment `ref e1 = ref e2`, the *ref-safe-to-escape* of `e2` must be at least as wide a scope as the *ref-safe-to-escape* of `e1`.
- For a ref return statement `return ref e1`, the *ref-safe-to-escape* of `e1` must be *ref-safe-to-escape* from the entire method. (TODO: Do we also need a rule that `e1` must be *safe-to-escape* from the entire method, or is that redundant?)
- For a return statement `return e1`, the *safe-to-escape* of `e1` must be *safe-to-escape* from the entire method.
- For an assignment `e1 = e2`, if the type of `e1` is a `ref struct` type, then the *safe-to-escape* of `e2` must be at least as wide a scope as the *safe-to-escape* of `e1`.
- For a method invocation if there is a `ref` or `out` argument of a `ref struct` type (including the receiver unless the type is `readonly`), with *safe-to-escape* E1, then no argument (including the receiver) may have a narrower *safe-to-escape* than E1. [Sample](#)
- A local function or anonymous function may not refer to a local or parameter of `ref struct` type declared in an enclosing scope.

Open Issue: We need some rule that permits us to produce an error when needing to spill a stack value of a `ref struct` type at an await expression, for example in the code

```
Foo(new Span<int>(...), await e2);
```

Explanations

These explanations and samples help explain why many of the safety rules above exist

Method Arguments Must Match

When invoking a method where there is an `out` or `ref` parameter that is a `ref struct` then all of the `ref struct` parameters need to have the same lifetime. This is necessary because C# must make all of its decisions around lifetime safety based on the information available in the signature of the method and the lifetime of the values at the call site.

When there are `ref` parameters that are `ref struct` then there is the potential that they could swap around their contents. Hence at the call site we must ensure all of these **potential** swaps are compatible. If the language didn't enforce that then it will allow for bad code like the following.

```

void M1(ref Span<int> s1)
{
    Span<int> s2 = stackalloc int[1];
    Swap(ref s1, ref s2);
}

void Swap(ref Span<int> x, ref Span<int> y)
{
    // This will effectively assign the stackalloc to the s1 parameter and allow it
    // to escape to the caller of M1
    ref x = ref y;
}

```

This analysis of `ref` parameters includes the receiver in instance methods. This is necessary because it can be used to store values passed in as parameters, just as a `ref` parameter could. This means with mismatched lifetimes you could create a type safety hole in the following way:

```

ref struct S
{
    public Span<int> Span;

    public void Set(Span<int> span)
    {
        Span = span;
    }
}

void Broken(ref S s)
{
    Span<int> span = stackalloc int[1];

    // The result of a stackalloc is now stored in s.Span and escaped to the caller
    // of Broken
    s.Set(span);
}

```

For the purpose of this analysis the receiver is considered an `in`, not a `ref`, if the type is a `readonly struct`. In that case the receiver cannot be used to store values from other parameters, it is effectively an `in` parameter for analysis purposes. Hence the same example above is legal when `S` is `readonly` because the `span` cannot be stored anywhere.

Struct This Escape

When it comes to span safety rules, the `this` value in an instance member is modeled as a parameter to the member. Now for a `struct` the type of `this` is actually `ref S` where in a `class` it's simply `S` (for members of a `class / struct` named `S`).

Yet `this` has different escaping rules than other `ref` parameters. Specifically it is not ref-safe-to-escape while other parameters are:

```

ref struct S
{
    int Field;

    // Illegal because `this` isn't safe to escape as ref
    ref int Get() => ref Field;

    // Legal
    ref int GetParam(ref int p) => ref p;
}

```

The reason for this restriction actually has little to do with `struct` member invocation. There are some rules that need to be worked out with respect to member invocation on `struct` members where the receiver is an rvalue. But that is very approachable.

The reason for this restriction is actually about interface invocation. Specifically it comes down to whether or not the following sample should or should not compile;

```
interface I1
{
    ref int Get();
}

ref int Use<T>(T p)
    where T : I1
{
    return ref p.Get();
}
```

Consider the case where `T` is instantiated as a `struct`. If the `this` parameter is ref-safe-to-escape then the return of `p.Get` could point to the stack (specifically it could be a field inside of the instantiated type of `T`). That means the language could not allow this sample to compile as it could be returning a `ref` to a stack location. On the other hand if `this` is not ref-safe-to-escape then `p.Get` cannot refer to the stack and hence it's safe to return.

This is why the escapability of `this` in a `struct` is really all about interfaces. It can absolutely be made to work but it has a trade off. The design eventually came down in favor of making interfaces more flexible.

There is potential for us to relax this in the future though.

Future Considerations

Length one `Span<T>` over ref values

Though not legal today there are cases where creating a length one `Span<T>` instance over a value would be beneficial:

```
void RefExample()
{
    int x = ...;

    // Today creating a length one Span<int> requires a stackalloc and a new
    // local
    Span<int> span1 = stackalloc [ ] { x };
    Use(span1);
    x = span1[0];

    // Simpler to just allow length one span
    var span2 = new Span<int>(ref x);
    Use(span2);
}
```

This feature gets more compelling if we lift the restrictions on [fixed sized buffers](#) as it would allow for `Span<T>` instances of even greater length.

If there is ever a need to go down this path then the language could accommodate this by ensuring such `Span<T>` instances were downward facing only. That is they were only ever *safe-to-escape* to the scope in which they were created. This ensure the language never had to consider a `ref` value escaping a method via a `ref struct` return or field of `ref struct`. This would likely also require further changes to recognize such

constructors as capturing a `ref` parameter in this way though.

Non-trailing named arguments

12/28/2021 • 3 minutes to read • [Edit Online](#)

Summary

Allow named arguments to be used in non-trailing position, as long as they are used in their correct position.

For example: `DoSomething(isEmployed:true, name, age);`.

Motivation

The main motivation is to avoid typing redundant information. It is common to name an argument that is a literal (such as `null`, `true`) for the purpose of clarifying the code, rather than of passing arguments out-of-order. That is currently disallowed (`CS1738`) unless all the following arguments are also named.

```
DoSomething(isEmployed:true, name, age); // currently disallowed, even though all arguments are in position
// CS1738 "Named argument specifications must appear after all fixed arguments have been specified"
```

Some additional examples:

```
public void DoSomething(bool isEmployed, string personName, int personAge) { ... }

DoSomething(isEmployed:true, name, age); // currently CS1738, but would become legal
DoSomething(true, personName:name, age); // currently CS1738, but would become legal
DoSomething(name, isEmployed:true, age); // remains illegal
DoSomething(name, age, isEmployed:true); // remains illegal
DoSomething(true, personAge:age, personName:name); // already legal
```

This would also work with params:

```
public class Task
{
    public static Task When(TaskStatus all, TaskStatus any, params Task[] tasks);
}

Task.When(all: TaskStatus.RanToCompletion, any: TaskStatus.Faulted, task1, task2)
```

Detailed design

In §7.5.1 (Argument lists), the spec currently says:

An *argument* with an *argument-name* is referred to as a **named argument**, whereas an *argument* without an *argument-name* is a **positional argument**. It is an error for a positional argument to appear after a named argument in an *argument-list*.

The proposal is to remove this error and update the rules for finding the corresponding parameter for an argument (§7.5.1.1):

Arguments in the argument-list of instance constructors, methods, indexers and delegates:

- [existing rules]
- An unnamed argument corresponds to no parameter when it is after an out-of-position named argument or

a named params argument.

In particular, this prevents invoking `void M(bool a = true, bool b = true, bool c = true,);` with `M(c: false, valueB);`. The first argument is used out-of-position (the argument is used in first position, but the parameter named "c" is in third position), so the following arguments should be named.

In other words, non-trailing named arguments are only allowed when the name and the position result in finding the same corresponding parameter.

Drawbacks

This proposal exacerbates existing subtleties with named arguments in overload resolution. For instance:

```
void M(int x, int y) { }
void M<T>(T y, int x) { }

void M2()
{
    M(3, 4);
    M(y: 3, x: 4); // Invokes M(int, int)
    M(y: 3, 4); // Invokes M<T>(T, int)
}
```

You could get this situation today by swapping the parameters:

```
void M(int y, int x) { }
void M<T>(int x, T y) { }

void M2()
{
    M(3, 4);
    M(x: 3, y: 4); // Invokes M(int, int)
    M(3, y: 4); // Invokes M<T>(int, T)
}
```

Similarly, if you have two methods `void M(int a, int b)` and `void M(int x, string y)`, the mistaken invocation `M(x: 1, 2)` will produce a diagnostic based on the second overload ("cannot convert from 'int' to 'string'"). This problem already exists when the named argument is used in a trailing position.

Alternatives

There are a couple of alternatives to consider:

- The status quo
- Providing IDE assistance to fill-in all the names of trailing arguments when you type specific a name in the middle.

Both of those suffer from more verbosity, as they introduce multiple named arguments even if you just need one name of a literal at the beginning of the argument list.

Unresolved questions

Design meetings

The feature was briefly discussed in LDM on May 16th 2017, with approval in principle (ok to move to proposal/prototype). It was also briefly discussed on June 28th 2017.

Relates to initial discussion <https://github.com/dotnet/csharp-lang/issues/518> Relates to championed issue <https://github.com/dotnet/csharp-lang/issues/570>

private protected

12/28/2021 • 9 minutes to read • [Edit Online](#)

Summary

Expose the CLR `protectedAndInternal` accessibility level in C# as `private protected`.

Motivation

There are many circumstances in which an API contains members that are only intended to be implemented and used by subclasses contained in the assembly that provides the type. While the CLR provides an accessibility level for that purpose, it is not available in C#. Consequently API owners are forced to either use `internal` protection and self-discipline or a custom analyzer, or to use `protected` with additional documentation explaining that, while the member appears in the public documentation for the type, it is not intended to be part of the public API. For examples of the latter, see members of Roslyn's `CSharpCompilationOptions` whose names start with `Common`.

Directly providing support for this access level in C# enables these circumstances to be expressed naturally in the language.

Detailed design

`private protected` **access modifier**

We propose to add a new access modifier combination `private protected` (which can appear in any order among the modifiers). This maps to the CLR notion of `protectedAndInternal`, and borrows the same syntax currently used in C++/CLI.

A member declared `private protected` can be accessed within a subclass of its container if that subclass is in the same assembly as the member.

We modify the language specification as follows (additions in bold). Section numbers are not shown below as they may vary depending on which version of the specification it is integrated into.

The declared accessibility of a member can be one of the following:

- Public, which is selected by including a public modifier in the member declaration. The intuitive meaning of public is “access not limited”.
 - Protected, which is selected by including a protected modifier in the member declaration. The intuitive meaning of protected is “access limited to the containing class or types derived from the containing class”.
 - Internal, which is selected by including an internal modifier in the member declaration. The intuitive meaning of internal is “access limited to this assembly”.
 - Protected internal, which is selected by including both a protected and an internal modifier in the member declaration. The intuitive meaning of protected internal is “accessible within this assembly as well as types derived from the containing class”.
 - **Private protected, which is selected by including both a private and a protected modifier in the member declaration. The intuitive meaning of private protected is “accessible within this assembly by types derived from the containing class”.**
-

Depending on the context in which a member declaration takes place, only certain types of declared accessibility are permitted. Furthermore, when a member declaration does not include any access modifiers, the context in which the declaration takes place determines the default declared accessibility.

- Namespaces implicitly have public declared accessibility. No access modifiers are allowed on namespace declarations.
- Types declared directly in compilation units or namespaces (as opposed to within other types) can have public or internal declared accessibility and default to internal declared accessibility.
- Class members can have any of the five kinds of declared accessibility and default to private declared accessibility. [Note: A type declared as a member of a class can have any of the five kinds of declared accessibility, whereas a type declared as a member of a namespace can have only public or internal declared accessibility. end note]
- Struct members can have public, internal, or private declared accessibility and default to private declared accessibility because structs are implicitly sealed. Struct members introduced in a struct (that is, not inherited by that struct) cannot have protected*, * or protected internal**, or private protected** declared accessibility. [Note: A type declared as a member of a struct can have public, internal, or private declared accessibility, whereas a type declared as a member of a namespace can have only public or internal declared accessibility. end note]
- Interface members implicitly have public declared accessibility. No access modifiers are allowed on interface member declarations.
- Enumeration members implicitly have public declared accessibility. No access modifiers are allowed on enumeration member declarations.

The accessibility domain of a nested member M declared in a type T within a program P, is defined as follows (noting that M itself might possibly be a type):

- If the declared accessibility of M is public, the accessibility domain of M is the accessibility domain of T.
- If the declared accessibility of M is protected internal, let D be the union of the program text of P and the program text of any type derived from T, which is declared outside P. The accessibility domain of M is the intersection of the accessibility domain of T with D.
- **If the declared accessibility of M is private protected, let D be the intersection of the program text of P and the program text of any type derived from T. The accessibility domain of M is the intersection of the accessibility domain of T with D.**
- If the declared accessibility of M is protected, let D be the union of the program text of T and the program text of any type derived from T. The accessibility domain of M is the intersection of the accessibility domain of T with D.
- If the declared accessibility of M is internal, the accessibility domain of M is the intersection of the accessibility domain of T with the program text of P.
- If the declared accessibility of M is private, the accessibility domain of M is the program text of T.

When a protected or **private protected** instance member is accessed outside the program text of the class in which it is declared, and when a protected internal instance member is accessed outside the program text of the program in which it is declared, the access shall take place within a class declaration that derives from the class in which it is declared. Furthermore, the access is required to take place through an instance of that derived class type or a class type constructed from it. This restriction prevents one derived class from accessing protected members of other derived classes, even when the members are inherited from the same base class.

The permitted access modifiers and the default access for a type declaration depend on the context in which the declaration takes place (§9.5.2):

- Types declared in compilation units or namespaces can have public or internal access. The default is internal access.
- Types declared in classes can have public, protected internal, **private protected**, protected, internal, or private access. The default is private access.
- Types declared in structs can have public, internal, or private access. The default is private access.

A static class declaration is subject to the following restrictions:

- A static class shall not include a sealed or abstract modifier. (However, since a static class cannot be instantiated or derived from, it behaves as if it was both sealed and abstract.)
- A static class shall not include a class-base specification (§16.2.5) and cannot explicitly specify a base class or a list of implemented interfaces. A static class implicitly inherits from type object.
- A static class shall only contain static members (§16.4.8). [Note: All constants and nested types are classified as static members. end note]
- A static class shall not have members with protected**, private protected** or protected internal declared accessibility.

It is a compile-time error to violate any of these restrictions.

A class-member-declaration can have any one of the ~~five~~**six** possible kinds of declared accessibility (§9.5.2): public, **private protected**, protected internal, protected, internal, or private. Except for the protected internal **and private protected** combinations, it is a compile-time error to specify more than one access modifier. When a class-member-declaration does not include any access modifiers, private is assumed.

Non-nested types can have public or internal declared accessibility and have internal declared accessibility by default. Nested types can have these forms of declared accessibility too, plus one or more additional forms of declared accessibility, depending on whether the containing type is a class or struct:

- A nested type that is declared in a class can have any of ~~five~~**six** forms of declared accessibility (public, **private protected**, protected internal, protected, internal, or private) and, like other class members, defaults to private declared accessibility.
- A nested type that is declared in a struct can have any of three forms of declared accessibility (public, internal, or private) and, like other struct members, defaults to private declared accessibility.

The method overridden by an override declaration is known as the overridden base method For an override method M declared in a class C, the overridden base method is determined by examining each base class type of C, starting with the direct base class type of C and continuing with each successive direct base class type, until in a given base class type at least one accessible method is located which has the same signature as M after substitution of type arguments. For the purposes of locating the overridden base method, a method is considered accessible if it is public, if it is protected, if it is protected internal, or if it is **either internal or private protected** and declared in the same program as C.

The use of accessor-modifiers is governed by the following restrictions:

- An accessor-modifier shall not be used in an interface or in an explicit interface member implementation.
- For a property or indexer that has no override modifier, an accessor-modifier is permitted only if the property or indexer has both a get and set accessor, and then is permitted only on one of those accessors.
- For a property or indexer that includes an override modifier, an accessor shall match the accessor-modifier, if any, of the accessor being overridden.
- The accessor-modifier shall declare an accessibility that is strictly more restrictive than the declared accessibility of the property or indexer itself. To be precise:
 - If the property or indexer has a declared accessibility of public, the accessor-modifier may be either **private protected**, , protected internal, internal, protected, or private.
 - If the property or indexer has a declared accessibility of protected internal, the accessor-modifier may be either **private protected**, internal, protected, or private.
 - If the property or indexer has a declared accessibility of internal or protected, the accessor-modifier shall be either **private protected** or private.
 - If the property or indexer has a declared accessibility of **private protected**, the accessor-modifier shall be private.
 - If the property or indexer has a declared accessibility of private, no accessor-modifier may be used.

Since inheritance isn't supported for structs, the declared accessibility of a struct member cannot be protected, **private protected**, or protected internal.

Drawbacks

As with any language feature, we must question whether the additional complexity to the language is repaid in the additional clarity offered to the body of C# programs that would benefit from the feature.

Alternatives

An alternative would be the provision of an API combining an attribute and an analyzer. The attribute is placed by the programmer on an `internal` member to indicate that the member is intended to be used only in subclasses, and the analyzer checks that those restrictions are obeyed.

Unresolved questions

The implementation is largely complete. The only open work item is drafting a corresponding specification for VB.

Design meetings

TBD

Conditional ref expressions

12/28/2021 • 2 minutes to read • [Edit Online](#)

The pattern of binding a ref variable to one or another expression conditionally is not currently expressible in C#.

The typical workaround is to introduce a method like:

```
ref T Choice(bool condition, ref T consequence, ref T alternative)
{
    if (condition)
    {
        return ref consequence;
    }
    else
    {
        return ref alternative;
    }
}
```

Note that this is not an exact replacement of a ternary since all arguments must be evaluated at the call site.

The following will not work as expected:

```
// will crash with NRE because 'arr[0]' will be executed unconditionally
ref var r = ref Choice(arr != null, ref arr[0], ref otherArr[0]);
```

The proposed syntax would look like:

```
<condition> ? ref <consequence> : ref <alternative>;
```

The above attempt with "Choice" can be *correctly* written using ref ternary as:

```
ref var r = ref (arr != null ? ref arr[0]: ref otherArr[0]);
```

The difference from Choice is that consequence and alternative expressions are accessed in a *truly* conditional manner, so we do not see a crash if `arr == null`

The ternary ref is just a ternary where both alternative and consequence are refs. It will naturally require that consequence/alternative operands are LValues. It will also require that consequence and alternative have types that are identity convertible to each other.

The type of the expression will be computed similarly to the one for the regular ternary. I.E. in a case if consequence and alternative have identity convertible, but different types, the existing type-merging rules will apply.

Safe-to-return will be assumed conservatively from the conditional operands. If either is unsafe to return the whole thing is unsafe to return.

Ref ternary is an LValue and as such it can be passed/assigned/returned by reference;

```
// pass by reference
foo(ref (arr != null ? ref arr[0]: ref otherArr[0]));

// return by reference
return ref (arr != null ? ref arr[0]: ref otherArr[0]);
```

Being an LValue, it can also be assigned to.

```
// assign to
(arr != null ? ref arr[0]: ref otherArr[0]) = 1;
```

Ref ternary can be used in a regular (not ref) context as well. Although it would not be common since you could as well just use a regular ternary.

```
int x = (arr != null ? ref arr[0]: ref otherArr[0]);
```

Implementation notes:

The complexity of the implementation would seem to be the size of a moderate-to-large bug fix. - I.E not very expensive. I do not think we need any changes to the syntax or parsing. There is no effect on metadata or interop. The feature is completely expression based. No effect on debugging/PDB either

Allow digit separator after 0b or 0x

12/28/2021 • 2 minutes to read • [Edit Online](#)

In C# 7.2, we extend the set of places that digit separators (the underscore character) can appear in integral literals. [Beginning in C# 7.0, separators are permitted between the digits of a literal.](#) Now, in C# 7.2, we also permit digit separators before the first significant digit of a binary or hexadecimal literal, after the prefix.

```
123      // permitted in C# 1.0 and later
1_2_3    // permitted in C# 7.0 and later
0x1_2_3  // permitted in C# 7.0 and later
0b101    // binary literals added in C# 7.0
0b1_0_1  // permitted in C# 7.0 and later

// in C# 7.2, _ is permitted after the `0x` or `0b`
0x_1_2   // permitted in C# 7.2 and later
0b_1_0_1 // permitted in C# 7.2 and later
```

We do not permit a decimal integer literal to have a leading underscore. A token such as `_123` is an identifier.

Unmanaged type constraint

12/28/2021 • 4 minutes to read • [Edit Online](#)

Summary

The unmanaged constraint feature will give language enforcement to the class of types known as "unmanaged types" in the C# language spec. This is defined in section 18.2 as a type which is not a reference type and doesn't contain reference type fields at any level of nesting.

Motivation

The primary motivation is to make it easier to author low level interop code in C#. Unmanaged types are one of the core building blocks for interop code, yet the lack of support in generics makes it impossible to create re-usable routines across all unmanaged types. Instead developers are forced to author the same boiler plate code for every unmanaged type in their library:

```
int Hash(Point point) { ... }
int Hash(TimeSpan timeSpan) { ... }
```

To enable this type of scenario the language will be introducing a new constraint: unmanaged:

```
void Hash<T>(T value) where T : unmanaged
{
    ...
}
```

This constraint can only be met by types which fit into the unmanaged type definition in the C# language spec. Another way of looking at it is that a type satisfies the unmanaged constraint if it can also be used as a pointer.

```
Hash(new Point()); // Okay
Hash(42); // Okay
Hash("hello") // Error: Type string does not satisfy the unmanaged constraint
```

Type parameters with the unmanaged constraint can use all the features available to unmanaged types: pointers, fixed, etc ...

```
void Hash<T>(T value) where T : unmanaged
{
    // Okay
    fixed (T* p = &value)
    {
        ...
    }
}
```

This constraint will also make it possible to have efficient conversions between structured data and streams of bytes. This is an operation that is common in networking stacks and serialization layers:


```
Span<byte> Convert<T>(ref T value) where T : unmanaged
{
    ...
}
```

Such routines are advantageous because they are provably safe at compile time and allocation free. Interop authors today can not do this (even though it's at a layer where perf is critical). Instead they need to rely on allocating routines that have expensive runtime checks to verify values are correctly unmanaged.

Detailed design

The language will introduce a new constraint named `unmanaged`. In order to satisfy this constraint a type must be a struct and all the fields of the type must fall into one of the following categories:

- Have the type `sbyte`, `byte`, `short`, `ushort`, `int`, `uint`, `long`, `ulong`, `char`, `float`, `double`, `decimal`, `bool`, `IntPtr` or `UIntPtr`.
- Be any `enum` type.
- Be a pointer type.
- Be a user defined struct that satisfies the `unmanaged` constraint.

Compiler generated instance fields, such as those backing auto-implemented properties, must also meet these constraints.

For example:

```
// Unmanaged type
struct Point
{
    int X;
    int Y {get; set;}
}

// Not an unmanaged type
struct Student
{
    string FirstName;
    string LastName;
}
```

The `unmanaged` constraint cannot be combined with `struct`, `class` or `new()`. This restriction derives from the fact that `unmanaged` implies `struct` hence the other constraints do not make sense.

The `unmanaged` constraint is not enforced by CLR, only by the language. To prevent mis-use by other languages, methods which have this constraint will be protected by a mod-req. This will prevent other languages from using type arguments which are not unmanaged types.

The token `unmanaged` in the constraint is not a keyword, nor a contextual keyword. Instead it is like `var` in that it is evaluated at that location and will either:

- Bind to user defined or referenced type named `unmanaged`: This will be treated just as any other named type constraint is treated.
- Bind to no type: This will be interpreted as the `unmanaged` constraint.

In the case there is a type named `unmanaged` and it is available without qualification in the current context, then there will be no way to use the `unmanaged` constraint. This parallels the rules surrounding the feature `var` and user defined types of the same name.

Drawbacks

The primary drawback of this feature is that it serves a small number of developers: typically low level library authors or frameworks. Hence it's spending precious language time for a small number of developers.

Yet these frameworks are often the basis for the majority of .NET applications out there. Hence performance / correctness wins at this level can have a ripple effect on the .NET ecosystem. This makes the feature worth considering even with the limited audience.

Alternatives

There are a couple of alternatives to consider:

- The status quo: The feature is not justified on its own merits and developers continue to use the implicit opt in behavior.

Questions

Metadata Representation

The F# language encodes the constraint in the signature file which means C# cannot re-use their representation. A new attribute will need to be chosen for this constraint. Additionally a method which has this constraint must be protected by a mod-req.

Blittable vs. Unmanaged

The F# language has a very [similar feature](#) which uses the keyword unmanaged. The blittable name comes from the use in Midori. May want to look to precedence here and use unmanaged instead.

Resolution The language decide to use unmanaged

Verifier

Does the verifier / runtime need to be updated to understand the use of pointers to generic type parameters? Or can it simply work as is without changes?

Resolution No changes needed. All pointer types are simply unverifiable.

Design meetings

n/a

Indexing `fixed` fields should not require pinning regardless of the movable/unmovable context.

12/28/2021 • 2 minutes to read • [Edit Online](#)

The change has the size of a bug fix. It can be in 7.3 and does not conflict with whatever direction we take further. This change is only about allowing the following scenario to work even though `s` is moveable. It is already valid when `s` is not moveable.

NOTE: in either case, it still requires `unsafe` context. It is possible to read uninitialized data or even out of range. That is not changing.

```
unsafe struct S
{
    public fixed int myFixedField[10];
}

class Program
{
    static S s;

    unsafe static void Main()
    {
        int p = s.myFixedField[5]; // indexing fixed-size array fields would be ok
    }
}
```

The main “challenge” that I see here is how to explain the relaxation in the spec. In particular, since the following would still need pinning. (because `s` is moveable and we explicitly use the field as a pointer)

```
unsafe struct S
{
    public fixed int myFixedField[10];
}

class Program
{
    static S s;

    unsafe static void Main()
    {
        int* ptr = s.myFixedField; // taking a pointer explicitly still requires pinning.
        int p = ptr[5];
    }
}
```

One reason why we require pinning of the target when it is movable is the artifact of our code generation strategy, - we always convert to an unmanaged pointer and thus force the user to pin via `fixed` statement. However, conversion to unmanaged is unnecessary when doing indexing. The same unsafe pointer math is equally applicable when we have the receiver in the form of a managed pointer. If we do that, then the intermediate ref is managed (GC-tracked) and the pinning is unnecessary.

The change <https://github.com/dotnet/roslyn/pull/24966> is a prototype PR that relaxes this requirement.

Pattern-based `fixed` statement

12/28/2021 • 3 minutes to read • [Edit Online](#)

Summary

Introduce a pattern that would allow types to participate in `fixed` statements.

Motivation

The language provides a mechanism for pinning managed data and obtain a native pointer to the underlying buffer.

```
fixed(byte* ptr = byteArray)
{
    // ptr is a native pointer to the first element of the array
    // byteArray is protected from being moved/collected by the GC for the duration of this block
}
```

The set of types that can participate in `fixed` is hardcoded and limited to arrays and `System.String`. Hardcoding "special" types does not scale when new primitives such as `ImmutableArray<T>`, `Span<T>`, `Utf8String` are introduced.

In addition, the current solution for `System.String` relies on a fairly rigid API. The shape of the API implies that `System.String` is a contiguous object that embeds UTF16 encoded data at a fixed offset from the object header. Such approach has been found problematic in several proposals that could require changes to the underlying layout. It would be desirable to be able to switch to something more flexible that decouples `System.String` object from its internal representation for the purpose of unmanaged interop.

Detailed design

Pattern

A viable pattern-based "fixed" need to:

- Provide the managed references to pin the instance and to initialize the pointer (preferably this is the same reference)
- Convey unambiguously the type of the unmanaged element (i.e. "char" for "string")
- Prescribe the behavior in "empty" case when there is nothing to refer to.
- Should not push API authors toward design decisions that hurt the use of the type outside of `fixed`.

I think the above could be satisfied by recognizing a specially named ref-returning member:

```
ref [readonly] T GetPinnableReference() .
```

In order to be used by the `fixed` statement the following conditions must be met:

1. There is only one such member provided for a type.
2. Returns by `ref` or `ref readonly`. (`readonly` is permitted so that authors of immutable/readonly types could implement the pattern without adding writeable API that could be used in safe code)
3. T is an unmanaged type. (since `T*` becomes the pointer type. The restriction will naturally expand if/when

the notion of "unmanaged" is expanded)

- Returns managed `nullptr` when there is no data to pin – probably the cheapest way to convey emptiness. (note that `""` string returns a ref to `'\0'` since strings are null-terminated)

Alternatively for the `#3` we can allow the result in empty cases be undefined or implementation-specific. That, however, may make the API more dangerous and prone to abuse and unintended compatibility burdens.

Translation

```
fixed(byte* ptr = thing)
{
    // <BODY>
}
```

becomes the following pseudocode (not all expressible in C#)

```
byte* ptr;
// specially decorated "pinned" IL local slot, not visible to user code.
pinned ref byte _pinned;

try
{
    // NOTE: null check is omitted for value types
    // NOTE: `thing` is evaluated only once (temporary is introduced if necessary)
    if (thing != null)
    {
        // obtain and "pin" the reference
        _pinned = ref thing.GetPinnableReference();

        // unsafe cast in IL
        ptr = (byte*)_pinned;
    }
    else
    {
        ptr = default(byte*);
    }

    // <BODY>
}
finally // finally can be omitted when not observable
{
    // "unpin" the object
    _pinned = nullptr;
}
```

Drawbacks

- GetPinnableReference is intended to be used only in `fixed`, but nothing prevents its use in safe code, so implementor must keep that in mind.

Alternatives

Users can introduce GetPinnableReference or similar member and use it as

```
fixed(byte* ptr = thing.GetPinnableReference())
{
    // <BODY>
}
```

There is no solution for `System.String` if alternative solution is desired.

Unresolved questions

- ☐ Behavior in "empty" state. - `nullptr` or `undefined` ?
- ☐ Should the extension methods be considered ?
- ☐ If a pattern is detected on `System.String`, should it win over ?

Design meetings

None yet.

Ref Local Reassignment

12/28/2021 • 2 minutes to read • [Edit Online](#)

In C# 7.3, we add support for rebinding the referent of a ref local variable or a ref parameter.

We add the following to the set of `assignment_operator`s.

```
assignment_operator
: '=' 'ref'
;
```

The `=ref` operator is called the **ref assignment operator**. It is not a *compound assignment operator*. The left operand must be an expression that binds to a ref local variable, a ref parameter (other than `this`), or an out parameter. The right operand must be an expression that yields an lvalue designating a value of the same type as the left operand.

The right operand must be definitely assigned at the point of the ref assignment.

When the left operand binds to an `out` parameter, it is an error if that `out` parameter has not been definitely assigned at the beginning of the ref assignment operator.

If the left operand is a writeable ref (i.e. it designates anything other than a `ref readonly` local or `in` parameter), then the right operand must be a writeable lvalue.

The ref assignment operator yields an lvalue of the assigned type. It is writeable if the left operand is writeable (i.e. not `ref readonly` or `in`).

The safety rules for this operator are:

- For a ref reassignment `e1 = ref e2`, the *ref-safe-to-escape* of `e2` must be at least as wide a scope as the *ref-safe-to-escape* of `e1`.

Where *ref-safe-to-escape* is defined in [Safety for ref-like types](#)

Stackalloc array initializers

12/28/2021 • 2 minutes to read • [Edit Online](#)

Summary

Allow array initializer syntax to be used with `stackalloc`.

Motivation

Ordinary arrays can have their elements initialized at creation time. It seems reasonable to allow that in `stackalloc` case.

The question of why such syntax is not allowed with `stackalloc` arises fairly frequently. See, for example, [#1112](#)

Detailed design

Ordinary arrays can be created through the following syntax:

```
new int[3]
new int[3] { 1, 2, 3 }
new int[] { 1, 2, 3 }
new[] { 1, 2, 3 }
```

We should allow stack allocated arrays be created through:

```
stackalloc int[3]    // currently allowed
stackalloc int[3] { 1, 2, 3 }
stackalloc int[] { 1, 2, 3 }
stackalloc[] { 1, 2, 3 }
```

The semantics of all cases is roughly the same as with arrays.

For example: in the last case the element type is inferred from the initializer and must be an "unmanaged" type.

NOTE: the feature is not dependent on the target being a `Span<T>`. It is just as applicable in `T*` case, so it does not seem reasonable to predicate it on `Span<T>` case.

Translation

The naive implementation could just initialize the array right after creation through a series of element-wise assignments.

Similarly to the case with arrays, it might be possible and desirable to detect cases where all or most of the elements are blittable types and use more efficient techniques by copying over the pre-created state of all the constant elements.

Drawbacks

Alternatives

This is a convenience feature. It is possible to just do nothing.

Unresolved questions

Design meetings

None yet.

Auto-Implemented Property Field-Targeted Attributes

12/28/2021 • 2 minutes to read • [Edit Online](#)

Summary

This feature intends to allow developers to apply attributes directly to the backing fields of auto-implemented properties.

Motivation

Currently it is not possible to apply attributes to the backing fields of auto-implemented properties. In those cases where the developer must use a field-targeting attribute they are forced to declare the field manually and use the more verbose property syntax. Given that C# has always supported field-targeted attributes on the generated backing field for events it makes sense to extend the same functionality to their property kin.

Detailed design

In short, the following would be legal C# and not produce a warning:

```
[Serializable]
public class Foo
{
    [field: NonSerialized]
    public string MySecret { get; set; }
}
```

This would result in the field-targeted attributes being applied to the compiler-generated backing field:

```
[Serializable]
public class Foo
{
    [NonSerialized]
    private string _mySecretBackingField;

    public string MySecret
    {
        get { return _mySecretBackingField; }
        set { _mySecretBackingField = value; }
    }
}
```

As mentioned, this brings parity with event syntax from C# 1.0 as the following is already legal and behaves as expected:

```
[Serializable]
public class Foo
{
    [field: NonSerialized]
    public event EventHandler MyEvent;
}
```

Drawbacks

There are two potential drawbacks to implementing this change:

1. Attempting to apply an attribute to the field of an auto-implemented property produces a compiler warning that the attributes in that block will be ignored. If the compiler were changed to support those attributes they would be applied to the backing field on a subsequent recompilation which could alter the behavior of the program at runtime.
2. The compiler does not currently validate the `AttributeUsage` targets of the attributes when attempting to apply them to the field of the auto-implemented property. If the compiler were changed to support field-targeted attributes and the attribute in question cannot be applied to a field the compiler would emit an error instead of a warning, breaking the build.

Alternatives

Unresolved questions

Design meetings

Expression variables in initializers

12/28/2021 • 2 minutes to read • [Edit Online](#)

Summary

We extend the features introduced in C# 7 to permit expressions containing expression variables (out variable declarations and declaration patterns) in field initializers, property initializers, ctor-initializers, and query clauses.

Motivation

This completes a couple of the rough edges left in the C# language due to lack of time.

Detailed design

We remove the restriction preventing the declaration of expression variables (out variable declarations and declaration patterns) in a ctor-initializer. Such a declared variable is in scope throughout the body of the constructor.

We remove the restriction preventing the declaration of expression variables (out variable declarations and declaration patterns) in a field or property initializer. Such a declared variable is in scope throughout the initializing expression.

We remove the restriction preventing the declaration of expression variables (out variable declarations and declaration patterns) in a query expression clause that is translated into the body of a lambda. Such a declared variable is in scope throughout that expression of the query clause.

Drawbacks

None.

Alternatives

The appropriate scope for expression variables declared in these contexts is not obvious, and deserves further LDM discussion.

Unresolved questions

- [] What is the appropriate scope for these variables?

Design meetings

None.

Support for == and != on tuple types

12/28/2021 • 4 minutes to read • [Edit Online](#)

Allow expressions `t1 == t2` where `t1` and `t2` are tuple or nullable tuple types of same cardinality, and evaluate them roughly as `temp1.Item1 == temp2.Item1 && temp1.Item2 == temp2.Item2` (assuming `var temp1 = t1; var temp2 = t2;`).

Conversely it would allow `t1 != t2` and evaluate it as

```
temp1.Item1 != temp2.Item1 || temp1.Item2 != temp2.Item2 .
```

In the nullable case, additional checks for `temp1.HasValue` and `temp2.HasValue` are used. For instance,

`nullableT1 == nullableT2` evaluates as

```
temp1.HasValue == temp2.HasValue ? (temp1.HasValue ? ... : true) : false .
```

When an element-wise comparison returns a non-bool result (for instance, when a non-bool user-defined `operator ==` or `operator !=` is used, or in a dynamic comparison), then that result will be either converted to `bool` or run through `operator true` or `operator false` to get a `bool`. The tuple comparison always ends up returning a `bool`.

As of C# 7.2, such code produces an error (

error CS0019: Operator '==' cannot be applied to operands of type '(...)' and '(...)'), unless there is a user-defined `operator==`.

Details

When binding the `==` (or `!=`) operator, the existing rules are: (1) dynamic case, (2) overload resolution, and (3) fail. This proposal adds a tuple case between (1) and (2): if both operands of a comparison operator are tuples (have tuple types or are tuple literals) and have matching cardinality, then the comparison is performed element-wise. This tuple equality is also lifted onto nullable tuples.

Both operands (and, in the case of tuple literals, their elements) are evaluated in order from left to right. Each pair of elements is then used as operands to bind the operator `==` (or `!=`), recursively. Any elements with compile-time type `dynamic` cause an error. The results of those element-wise comparisons are used as operands in a chain of conditional AND (or OR) operators.

For instance, in the context of `(int, (int, int)) t1, t2; , t1 == (1, (2, 3))` would evaluate as

```
temp1.Item1 == temp2.Item1 && temp1.Item2.Item1 == temp2.Item2.Item1 && temp1.Item2.Item2 == temp2.Item2.Item2
```

When a tuple literal is used as operand (on either side), it receives a converted tuple type formed by the element-wise conversions which are introduced when binding the operator `==` (or `!=`) element-wise.

For instance, in `(1L, 2, "hello") == (1, 2L, null)`, the converted type for both tuple literals is `(long, long, string)` and the second literal has no natural type.

Deconstruction and conversions to tuple

In `(a, b) == x`, the fact that `x` can deconstruct into two elements does not play a role. That could conceivably be in a future proposal, although it would raise questions about `x == y` (is this a simple comparison or an element-wise comparison, and if so using what cardinality?). Similarly, conversions to tuple play no role.

Tuple element names

When converting a tuple literal, we warn when an explicit tuple element name was provided in the literal, but it doesn't match the target tuple element name. We use the same rule in tuple comparison, so that assuming

```
(int a, int b) t we warn on d in t == (c, d: 0) .
```

Non-bool element-wise comparison results

If an element-wise comparison is dynamic in a tuple equality, we use a dynamic invocation of the operator `false` and negate that to get a `bool` and continue with further element-wise comparisons.

If an element-wise comparison returns some other non-bool type in a tuple equality, there are two cases:

- if the non-bool type converts to `bool`, we apply that conversion,
- if there is no such conversion, but the type has an operator `false`, we'll use that and negate the result.

In a tuple inequality, the same rules apply except that we'll use the operator `true` (without negation) instead of the operator `false`.

Those rules are similar to the rules involved for using a non-bool type in an `if` statement and some other existing contexts.

Evaluation order and special cases

The left-hand-side value is evaluated first, then the right-hand-side value, then the element-wise comparisons from left to right (including conversions, and with early exit based on existing rules for conditional AND/OR operators).

For instance, if there is a conversion from type `A` to type `B` and a method `(A, A) GetTuple()`, evaluating `(new A(1), (new B(2), new B(3))) == (new B(4), GetTuple())` means:

- `new A(1)`
- `new B(2)`
- `new B(3)`
- `new B(4)`
- `GetTuple()`
- then the element-wise conversions and comparisons and conditional logic is evaluated (convert `new A(1)` to type `B`, then compare it with `new B(4)`, and so on).

Comparing `null` to `null`

This is a special case from regular comparisons, that carries over to tuple comparisons. The `null == null` comparison is allowed, and the `null` literals do not get any type. In tuple equality, this means, `(0, null) == (0, null)` is also allowed and the `null` and tuple literals don't get a type either.

Comparing a nullable struct to `null` without `operator==`

This is another special case from regular comparisons, that carries over to tuple comparisons. If you have a `struct S` without `operator==`, the `(S?)x == null` comparison is allowed, and it is interpreted as `((S?).x).HasValue`. In tuple equality, the same rule is applied, so `(0, (S?)x) == (0, null)` is allowed.

Compatibility

If someone wrote their own `ValueTuple` types with an implementation of the comparison operator, it would have previously been picked up by overload resolution. But since the new tuple case comes before overload resolution, we would handle this case with tuple comparison instead of relying on the user-defined comparison.

Improved overload candidates

12/28/2021 • 2 minutes to read • [Edit Online](#)

Summary

The overload resolution rules have been updated in nearly every C# language update to improve the experience for programmers, making ambiguous invocations select the "obvious" choice. This has to be done carefully to preserve backward compatibility, but since we are usually resolving what would otherwise be error cases, these enhancements usually work out nicely.

1. When a method group contains both instance and static members, we discard the instance members if invoked without an instance receiver or context, and discard the static members if invoked with an instance receiver. When there is no receiver, we include only static members in a static context, otherwise both static and instance members. When the receiver is ambiguously an instance or type due to a color-color situation, we include both. A static context, where an implicit this instance receiver cannot be used, includes the body of members where no this is defined, such as static members, as well as places where this cannot be used, such as field initializers and constructor-initializers.
2. When a method group contains some generic methods whose type arguments do not satisfy their constraints, these members are removed from the candidate set.
3. For a method group conversion, candidate methods whose return type doesn't match up with the delegate's return type are removed from the set.

Nullable reference types in C#

12/28/2021 • 7 minutes to read • [Edit Online](#)

The goal of this feature is to:

- Allow developers to express whether a variable, parameter or result of a reference type is intended to be null or not.
- Provide warnings when such variables, parameters and results are not used according to that intent.

Expression of intent

The language already contains the `T?` syntax for value types. It is straightforward to extend this syntax to reference types.

It is assumed that the intent of an unadorned reference type `T` is for it to be non-null.

Checking of nullable references

A flow analysis tracks nullable reference variables. Where the analysis deems that they would not be null (e.g. after a check or an assignment), their value will be considered a non-null reference.

A nullable reference can also explicitly be treated as non-null with the postfix `x!` operator (the "damnit" operator), for when flow analysis cannot establish a non-null situation that the developer knows is there.

Otherwise, a warning is given if a nullable reference is dereferenced, or is converted to a non-null type.

A warning is given when converting from `S[]` to `T?[]` and from `S?[]` to `T[]`.

A warning is given when converting from `C<S>` to `C<T?>` except when the type parameter is covariant (`out`), and when converting from `C<S?>` to `C<T>` except when the type parameter is contravariant (`in`).

A warning is given on `C<T?>` if the type parameter has non-null constraints.

Checking of non-null references

A warning is given if a null literal is assigned to a non-null variable or passed as a non-null parameter.

A warning is also given if a constructor does not explicitly initialize non-null reference fields.

We cannot adequately track that all elements of an array of non-null references are initialized. However, we could issue a warning if no element of a newly created array is assigned to before the array is read from or passed on. That might handle the common case without being too noisy.

We need to decide whether `default(T)` generates a warning, or is simply treated as being of the type `T?`.

Metadata representation

Nullability adornments should be represented in metadata as attributes. This means that downlevel compilers will ignore them.

We need to decide if only nullable annotations are included, or there's also some indication of whether non-null was "on" in the assembly.

Generics

If a type parameter `T` has non-nullable constraints, it is treated as non-nullable within its scope.

If a type parameter is unconstrained or has only nullable constraints, the situation is a little more complex: this means that the corresponding type argument could be *either* nullable or non-nullable. The safe thing to do in that situation is to treat the type parameter as *both* nullable and non-nullable, giving warnings when either is violated.

It is worth considering whether explicit nullable reference constraints should be allowed. Note, however, that we cannot avoid having nullable reference types *implicitly* be constraints in certain cases (inherited constraints).

The `class` constraint is non-null. We can consider whether `class?` should be a valid nullable constraint denoting "nullable reference type".

Type inference

In type inference, if a contributing type is a nullable reference type, the resulting type should be nullable. In other words, nullness is propagated.

We should consider whether the `null` literal as a participating expression should contribute nullness. It doesn't today: for value types it leads to an error, whereas for reference types the null successfully converts to the plain type.

```
string? n = "world";
var x = b ? "Hello" : n; // string?
var y = b ? "Hello" : null; // string? or error
var z = b ? 7 : null; // Error today, could be int?
```

Null guard guidance

As a feature, nullable reference types allow developers to express their intent, and provide warnings through flow analysis if that intent is contradicted. There is a common question as to whether or not null guards are necessary.

Example of null guard

```
public void DoWork(Worker worker)
{
    // Guard against worker being null
    if (worker is null)
    {
        throw new ArgumentNullException(nameof(worker));
    }

    // Otherwise use worker argument
}
```

In the previous example, the `DoWork` function accepts a `Worker` and guards against it potentially being `null`. If the `worker` argument is `null`, the `DoWork` function will `throw`. With nullable reference types, the code in the previous example makes the intent that the `Worker` parameter would *not* be `null`. If the `DoWork` function was a public API, such as a NuGet package or a shared library - as guidance you should leave null guards in place. As a public API, the only guarantee that a caller isn't passing `null` is to guard against it.

Express intent

A more compelling use of the previous example is to express that the `Worker` parameter could be `null`, thus

making the null guard more appropriate. If you remove the null guard in the following example, the compiler warns that you may be dereferencing null. Regardless, both null guards are still valid.

```
public void DoWork(Worker? worker)
{
    // Guard against worker being null
    if (worker is null)
    {
        throw new ArgumentNullException(nameof(worker));
    }

    // Otherwise use worker argument
}
```

For non-public APIs, such as source code entirely in control by a developer or dev team - the nullable reference types could allow for the safe removal of null guards where the developers can guarantee it is not necessary. The feature can help with warnings, but it cannot guarantee that at runtime code execution could result in a `NullReferenceException`.

Breaking changes

Non-null warnings are an obvious breaking change on existing code, and should be accompanied with an opt-in mechanism.

Less obviously, warnings from nullable types (as described above) are a breaking change on existing code in certain scenarios where the nullability is implicit:

- Unconstrained type parameters will be treated as implicitly nullable, so assigning them to `object` or accessing e.g. `ToString` will yield warnings.
- if type inference infers nullness from `null` expressions, then existing code will sometimes yield nullable rather than non-nullable types, which can lead to new warnings.

So nullable warnings also need to be optional

Finally, adding annotations to an existing API will be a breaking change to users who have opted in to warnings, when they upgrade the library. This, too, merits the ability to opt in or out. "I want the bug fixes, but I am not ready to deal with their new annotations"

In summary, you need to be able to opt in/out of:

- Nullable warnings
- Non-null warnings
- Warnings from annotations in other files

The granularity of the opt-in suggests an analyzer-like model, where swaths of code can opt in and out with pragmas and severity levels can be chosen by the user. Additionally, per-library options ("ignore the annotations from JSON.NET until I'm ready to deal with the fall out") may be expressible in code as attributes.

The design of the opt-in/transition experience is crucial to the success and usefulness of this feature. We need to make sure that:

- Users can adopt nullability checking gradually as they want to
- Library authors can add nullability annotations without fear of breaking customers
- Despite these, there is not a sense of "configuration nightmare"

Tweaks

We could consider not using the `?` annotations on locals, but just observing whether they are used in accordance with what gets assigned to them. I don't favor this; I think we should uniformly let people express their intent.

We could consider a shorthand `T! x` on parameters, that auto-generates a runtime null check.

Certain patterns on generic types, such as `FirstOrDefault` or `TryGet`, have slightly weird behavior with non-nullable type arguments, because they explicitly yield default values in certain situations. We could try to nuance the type system to accommodate these better. For instance, we could allow `?` on unconstrained type parameters, even though the type argument could already be nullable. I doubt that it is worth it, and it leads to weirdness related to interaction with nullable *value* types.

Nullable value types

We could consider adopting some of the above semantics for nullable value types as well.

We already mentioned type inference, where we could infer `int?` from `(7, null)`, instead of just giving an error.

Another opportunity is to apply the flow analysis to nullable value types. When they are deemed non-null, we could actually allow using as the non-nullable type in certain ways (e.g. member access). We just have to be careful that the things that you can *already* do on a nullable value type will be preferred, for back compat reasons.

Recursive Pattern Matching

12/28/2021 • 12 minutes to read • [Edit Online](#)

Summary

Pattern matching extensions for C# enable many of the benefits of algebraic data types and pattern matching from functional languages, but in a way that smoothly integrates with the feel of the underlying language. Elements of this approach are inspired by related features in the programming languages [F#](#) and [Scala](#).

Detailed design

Is Expression

The `is` operator is extended to test an expression against a *pattern*.

```
relational_expression
    : is_pattern_expression
    ;
is_pattern_expression
    : relational_expression 'is' pattern
    ;
```

This form of *relational_expression* is in addition to the existing forms in the C# specification. It is a compile-time error if the *relational_expression* to the left of the `is` token does not designate a value or does not have a type.

Every *identifier* of the pattern introduces a new local variable that is *definitely assigned* after the `is` operator is `true` (i.e. *definitely assigned when true*).

Note: There is technically an ambiguity between *type* in an `is-expression` and *constant_pattern*, either of which might be a valid parse of a qualified identifier. We try to bind it as a type for compatibility with previous versions of the language; only if that fails do we resolve it as we do an expression in other contexts, to the first thing found (which must be either a constant or a type). This ambiguity is only present on the right-hand-side of an `is` expression.

Patterns

Patterns are used in the *is_pattern* operator, in a *switch_statement*, and in a *switch_expression* to express the shape of data against which incoming data (which we call the input value) is to be compared. Patterns may be recursive so that parts of the data may be matched against sub-patterns.

```

pattern
  : declaration_pattern
  | constant_pattern
  | var_pattern
  | positional_pattern
  | property_pattern
  | discard_pattern
  ;
declaration_pattern
  : type simple_designation
  ;
constant_pattern
  : constant_expression
  ;
var_pattern
  : 'var' designation
  ;
positional_pattern
  : type? '(' subpatterns? ')' property_subpattern? simple_designation?
  ;
subpatterns
  : subpattern
  | subpattern ',' subpatterns
  ;
subpattern
  : pattern
  | identifier ':' pattern
  ;
property_subpattern
  : '{' '}'
  | '{' subpatterns ',' '?' '}'
  ;
property_pattern
  : type? property_subpattern simple_designation?
  ;
simple_designation
  : single_variable_designation
  | discard_designation
  ;
discard_pattern
  : '_'
  ;

```

Declaration Pattern

```

declaration_pattern
  : type simple_designation
  ;

```

The *declaration_pattern* both tests that an expression is of a given type and casts it to that type if the test succeeds. This may introduce a local variable of the given type named by the given identifier, if the designation is a *single_variable_designation*. That local variable is *definitely assigned* when the result of the pattern-matching operation is `true`.

The runtime semantic of this expression is that it tests the runtime type of the left-hand *relational_expression* operand against the *type* in the pattern. If it is of that runtime type (or some subtype) and not `null`, the result of the `is operator` is `true`.

Certain combinations of static type of the left-hand-side and the given type are considered incompatible and result in compile-time error. A value of static type `E` is said to be *pattern-compatible* with a type `T` if there exists an identity conversion, an implicit reference conversion, a boxing conversion, an explicit reference conversion, or an unboxing conversion from `E` to `T`, or if one of those types is an open type. It is a compile-

time error if an input of type `E` is not *pattern-compatible* with the *type* in a type pattern that it is matched with.

The type pattern is useful for performing run-time type tests of reference types, and replaces the idiom

```
var v = expr as Type;
if (v != null) { // code using v
```

With the slightly more concise

```
if (expr is Type v) { // code using v
```

It is an error if *type* is a nullable value type.

The type pattern can be used to test values of nullable types: a value of type `Nullable<T>` (or a boxed `T`) matches a type pattern `T2 id` if the value is non-null and the type of `T2` is `T`, or some base type or interface of `T`. For example, in the code fragment

```
int? x = 3;
if (x is int v) { // code using v
```

The condition of the `if` statement is `true` at runtime and the variable `v` holds the value `3` of type `int` inside the block. After the block the variable `v` is in scope but not definitely assigned.

Constant Pattern

```
constant_pattern
    : constant_expression
    ;
```

A constant pattern tests the value of an expression against a constant value. The constant may be any constant expression, such as a literal, the name of a declared `const` variable, or an enumeration constant. When the input value is not an open type, the constant expression is implicitly converted to the type of the matched expression; if the type of the input value is not *pattern-compatible* with the type of the constant expression, the pattern-matching operation is an error.

The pattern `c` is considered matching the converted input value `e` if `object.Equals(c, e)` would return `true`.

We expect to see `e is null` as the most common way to test for `null` in newly written code, as it cannot invoke a user-defined `operator==`.

Var Pattern

```

var_pattern
: 'var' designation
;
designation
: simple_designation
| tuple_designation
;
simple_designation
: single_variable_designation
| discard_designation
;
single_variable_designation
: identifier
;
discard_designation
: _
;
tuple_designation
: '(' designations? ')'
;
designations
: designation
| designations ',' designation
;

```

If the *designation* is a *simple_designation*, an expression *e* matches the pattern. In other words, a match to a *var pattern* always succeeds with a *simple_designation*. If the *simple_designation* is a *single_variable_designation*, the value of *e* is bounds to a newly introduced local variable. The type of the local variable is the static type of *e*.

If the *designation* is a *tuple_designation*, then the pattern is equivalent to a *positional_pattern* of the form `(var designation, ...)` where the *designations* are those found within the *tuple_designation*. For example, the pattern `var (x, (y, z))` is equivalent to `(var x, (var y, var z))`.

It is an error if the name `var` binds to a type.

Discard Pattern

```

discard_pattern
: '_'
;

```

An expression *e* matches the pattern `_` always. In other words, every expression matches the discard pattern.

A discard pattern may not be used as the pattern of an *is_pattern_expression*.

Positional Pattern

A positional pattern checks that the input value is not `null`, invokes an appropriate `Deconstruct` method, and performs further pattern matching on the resulting values. It also supports a tuple-like pattern syntax (without the type being provided) when the type of the input value is the same as the type containing `Deconstruct`, or if the type of the input value is a tuple type, or if the type of the input value is `object` or `ITuple` and the runtime type of the expression implements `ITuple`.

```

positional_pattern
  : type? '(' subpatterns? ')' property_subpattern? simple_designation?
  ;
subpatterns
  : subpattern
  | subpattern ',' subpatterns
  ;
subpattern
  : pattern
  | identifier ':' pattern
  ;

```

If the *type* is omitted, we take it to be the static type of the input value.

Given a match of an input value to the pattern *type* `(subpattern_list)`, a method is selected by searching in *type* for accessible declarations of `Deconstruct` and selecting one among them using the same rules as for the deconstruction declaration.

It is an error if a *positional_pattern* omits the type, has a single *subpattern* without an *identifier*, has no *property_subpattern* and has no *simple_designation*. This disambiguates between a *constant_pattern* that is parenthesized and a *positional_pattern*.

In order to extract the values to match against the patterns in the list,

- If *type* was omitted and the input value's type is a tuple type, then the number of subpatterns is required to be the same as the cardinality of the tuple. Each tuple element is matched against the corresponding *subpattern*, and the match succeeds if all of these succeed. If any *subpattern* has an *identifier*, then that must name a tuple element at the corresponding position in the tuple type.
- Otherwise, if a suitable `Deconstruct` exists as a member of *type*, it is a compile-time error if the type of the input value is not *pattern-compatible* with *type*. At runtime the input value is tested against *type*. If this fails then the positional pattern match fails. If it succeeds, the input value is converted to this type and `Deconstruct` is invoked with fresh compiler-generated variables to receive the `out` parameters. Each value that was received is matched against the corresponding *subpattern*, and the match succeeds if all of these succeed. If any *subpattern* has an *identifier*, then that must name a parameter at the corresponding position of `Deconstruct`.
- Otherwise if *type* was omitted, and the input value is of type `object` or `ITuple` or some type that can be converted to `ITuple` by an implicit reference conversion, and no *identifier* appears among the subpatterns, then we match using `ITuple`.
- Otherwise the pattern is a compile-time error.

The order in which subpatterns are matched at runtime is unspecified, and a failed match may not attempt to match all subpatterns.

Example

This example uses many of the features described in this specification

```

var newState = (GetState(), action, hasKey) switch {
    (DoorState.Closed, Action.Open, _) => DoorState.Opened,
    (DoorState.Opened, Action.Close, _) => DoorState.Closed,
    (DoorState.Closed, Action.Lock, true) => DoorState.Locked,
    (DoorState.Locked, Action.Unlock, true) => DoorState.Closed,
    (var state, _, _) => state };

```

Property Pattern

A property pattern checks that the input value is not `null` and recursively matches values extracted by the use of accessible properties or fields.


```

property_pattern
    : type? property_subpattern simple_designation?
    ;
property_subpattern
    : '{' '}'
    | '{' subpatterns ','? '}'
    ;

```

It is an error if any *subpattern* of a *property_pattern* does not contain an *identifier* (it must be of the second form, which has an *identifier*). A trailing comma after the last subpattern is optional.

Note that a null-checking pattern falls out of a trivial property pattern. To check if the string `s` is non-null, you can write any of the following forms

```

if (s is object o) ... // o is of type object
if (s is string x) ... // x is of type string
if (s is {} x) ... // x is of type string
if (s is {}) ...

```

Given a match of an expression *e* to the pattern `type { property_pattern_list }`, it is a compile-time error if the expression *e* is not *pattern-compatible* with the type *T* designated by *type*. If the type is absent, we take it to be the static type of *e*. If the *identifier* is present, it declares a pattern variable of type *type*. Each of the identifiers appearing on the left-hand-side of its *property_pattern_list* must designate an accessible readable property or field of *T*. If the *simple_designation* of the *property_pattern* is present, it defines a pattern variable of type *T*.

At runtime, the expression is tested against *T*. If this fails then the property pattern match fails and the result is `false`. If it succeeds, then each *property_subpattern* field or property is read and its value matched against its corresponding pattern. The result of the whole match is `false` only if the result of any of these is `false`. The order in which subpatterns are matched is not specified, and a failed match may not match all subpatterns at runtime. If the match succeeds and the *simple_designation* of the *property_pattern* is a *single_variable_designation*, it defines a variable of type *T* that is assigned the matched value.

Note: The property pattern can be used to pattern-match with anonymous types.

Example

```

if (o is string { Length: 5 } s)

```

Switch Expression

A *switch_expression* is added to support `switch`-like semantics for an expression context.

The C# language syntax is augmented with the following syntactic productions:

```

multiplicative_expression
: switch_expression
| multiplicative_expression '*' switch_expression
| multiplicative_expression '/' switch_expression
| multiplicative_expression '%' switch_expression
;
switch_expression
: range_expression 'switch' '{' '}'
| range_expression 'switch' '{' switch_expression_arms ',' '}'
;
switch_expression_arms
: switch_expression_arm
| switch_expression_arms ',' switch_expression_arm
;
switch_expression_arm
: pattern case_guard? '=>' expression
;
case_guard
: 'when' null_coalescing_expression
;

```

The *switch_expression* is not permitted as an *expression_statement*.

We are looking at relaxing this in a future revision.

The type of the *switch_expression* is the *best common type* of the expressions appearing to the right of the `=>` tokens of the *switch_expression_arms* if such a type exists and the expression in every arm of the switch expression can be implicitly converted to that type. In addition, we add a new *switch_expression_conversion*, which is a predefined implicit conversion from a switch expression to every type `T` for which there exists an implicit conversion from each arm's expression to `T`.

It is an error if some *switch_expression_arm*'s pattern cannot affect the result because some previous pattern and guard will always match.

A switch expression is said to be *exhaustive* if some arm of the switch expression handles every value of its input. The compiler shall produce a warning if a switch expression is not *exhaustive*.

At runtime, the result of the *switch_expression* is the value of the *expression* of the first *switch_expression_arm* for which the expression on the left-hand-side of the *switch_expression* matches the *switch_expression_arm*'s pattern, and for which the *case_guard* of the *switch_expression_arm*, if present, evaluates to `true`. If there is no such *switch_expression_arm*, the *switch_expression* throws an instance of the exception

`System.Runtime.CompilerServices.SwitchExpressionException`.

Optional parens when switching on a tuple literal

In order to switch on a tuple literal using the *switch_statement*, you have to write what appear to be redundant parens

```

switch ((a, b))
{

```

To permit

```

switch (a, b)
{

```

the parentheses of the switch statement are optional when the expression being switched on is a tuple literal.

Order of evaluation in pattern-matching

Giving the compiler flexibility in reordering the operations executed during pattern-matching can permit flexibility that can be used to improve the efficiency of pattern-matching. The (unenforced) requirement would be that properties accessed in a pattern, and the Deconstruct methods, are required to be "pure" (side-effect free, idempotent, etc). That doesn't mean that we would add purity as a language concept, only that we would allow the compiler flexibility in reordering operations.

Resolution 2018-04-04 LDM: confirmed: the compiler is permitted to reorder calls to `Deconstruct`, property accesses, and invocations of methods in `ITuple`, and may assume that returned values are the same from multiple calls. The compiler should not invoke functions that cannot affect the result, and we will be very careful before making any changes to the compiler-generated order of evaluation in the future.

Some Possible Optimizations

The compilation of pattern matching can take advantage of common parts of patterns. For example, if the top-level type test of two successive patterns in a *switch_statement* is the same type, the generated code can skip the type test for the second pattern.

When some of the patterns are integers or strings, the compiler can generate the same kind of code it generates for a switch-statement in earlier versions of the language.

For more on these kinds of optimizations, see [\[Scott and Ramsey \(2000\)\]](#).

default interface methods

12/28/2021 • 27 minutes to read • [Edit Online](#)

Summary

Add support for *virtual extension methods* - methods in interfaces with concrete implementations. A class or struct that implements such an interface is required to have a single *most specific* implementation for the interface method, either implemented by the class or struct, or inherited from its base classes or interfaces. Virtual extension methods enable an API author to add methods to an interface in future versions without breaking source or binary compatibility with existing implementations of that interface.

These are similar to Java's ["Default Methods"](#).

(Based on the likely implementation technique) this feature requires corresponding support in the CLI/CLR. Programs that take advantage of this feature cannot run on earlier versions of the platform.

Motivation

The principal motivations for this feature are

- Default interface methods enable an API author to add methods to an interface in future versions without breaking source or binary compatibility with existing implementations of that interface.
- The feature enables C# to interoperate with APIs targeting [Android \(Java\)](#) and [iOS \(Swift\)](#), which support similar features.
- As it turns out, adding default interface implementations provides the elements of the "traits" language feature ([https://en.wikipedia.org/wiki/Trait_\(computer_programming\)](https://en.wikipedia.org/wiki/Trait_(computer_programming))). Traits have proven to be a powerful programming technique (<http://scg.unibe.ch/archive/papers/Scha03aTraits.pdf>).

Detailed design

The syntax for an interface is extended to permit

- member declarations that declare constants, operators, static constructors, and nested types;
- a *body* for a method or indexer, property, or event accessor (that is, a "default" implementation);
- member declarations that declare static fields, methods, properties, indexers, and events;
- member declarations using the explicit interface implementation syntax; and
- Explicit access modifiers (the default access is `public`).

Members with bodies permit the interface to provide a "default" implementation for the method in classes and structs that do not provide an overriding implementation.

Interfaces may not contain instance state. While static fields are now permitted, instance fields are not permitted in interfaces. Instance auto-properties are not supported in interfaces, as they would implicitly declare a hidden field.

Static and private methods permit useful refactoring and organization of code used to implement the interface's public API.

A method override in an interface must use the explicit interface implementation syntax.

It is an error to declare a class type, struct type, or enum type within the scope of a type parameter that was declared with a *variance annotation*. For example, the declaration of `C` below is an error.

```
interface IOuter<out T>
{
    class C { } // error: class declaration within the scope of variant type parameter 'T'
}
```

Concrete methods in interfaces

The simplest form of this feature is the ability to declare a *concrete method* in an interface, which is a method with a body.

```
interface IA
{
    void M() { WriteLine("IA.M"); }
}
```

A class that implements this interface need not implement its concrete method.

```
class C : IA { } // OK

IA i = new C();
i.M(); // prints "IA.M"
```

The final override for `IA.M` in class `C` is the concrete method `M` declared in `IA`. Note that a class does not inherit members from its interfaces; that is not changed by this feature:

```
new C().M(); // error: class 'C' does not contain a member 'M'
```

Within an instance member of an interface, `this` has the type of the enclosing interface.

Modifiers in interfaces

The syntax for an interface is relaxed to permit modifiers on its members. The following are permitted: `private`, `protected`, `internal`, `public`, `virtual`, `abstract`, `sealed`, `static`, `extern`, and `partial`.

TODO: check what other modifiers exist.

An interface member whose declaration includes a body is a `virtual` member unless the `sealed` or `private` modifier is used. The `virtual` modifier may be used on a function member that would otherwise be implicitly `virtual`. Similarly, although `abstract` is the default on interface members without bodies, that modifier may be given explicitly. A non-virtual member may be declared using the `sealed` keyword.

It is an error for a `private` or `sealed` function member of an interface to have no body. A `private` function member may not have the modifier `sealed`.

Access modifiers may be used on interface members of all kinds of members that are permitted. The access level `public` is the default but it may be given explicitly.

Open Issue: We need to specify the precise meaning of the access modifiers such as `protected` and `internal`, and which declarations do and do not override them (in a derived interface) or implement them (in a class that implements the interface).

Interfaces may declare `static` members, including nested types, methods, indexers, properties, events, and static constructors. The default access level for all interface members is `public`.

Interfaces may not declare instance constructors, destructors, or fields.

Closed Issue: Should operator declarations be permitted in an interface? Probably not conversion operators, but what about others? **Decision:** Operators are permitted *except* for conversion, equality, and inequality operators.

Closed Issue: Should `new` be permitted on interface member declarations that hide members from base interfaces? **Decision:** Yes.

Closed Issue: We do not currently permit `partial` on an interface or its members. That would require a separate proposal. **Decision:** Yes. <https://github.com/dotnet/csharplang/blob/master/meetings/2018/LDM-2018-10-17.md#permit-partial-in-interface>

Overrides in interfaces

Override declarations (i.e. those containing the `override` modifier) allow the programmer to provide a most specific implementation of a virtual member in an interface where the compiler or runtime would not otherwise find one. It also allows turning an abstract member from a super-interface into a default member in a derived interface. An override declaration is permitted to *explicitly* override a particular base interface method by qualifying the declaration with the interface name (no access modifier is permitted in this case). Implicit overrides are not permitted.

```
interface IA
{
    void M() { WriteLine("IA.M"); }
}
interface IB : IA
{
    override void IA.M() { WriteLine("IB.M"); } // explicitly named
}
interface IC : IA
{
    override void M() { WriteLine("IC.M"); } // implicitly named
}
```

Override declarations in interfaces may not be declared `sealed`.

Public `virtual` function members in an interface may be overridden in a derived interface explicitly (by qualifying the name in the override declaration with the interface type that originally declared the method, and omitting an access modifier).

`virtual` function members in an interface may only be overridden explicitly (not implicitly) in derived interfaces, and members that are not `public` may only be implemented in a class or struct explicitly (not implicitly). In either case, the overridden or implemented member must be *accessible* where it is overridden.

Reabstraction

A virtual (concrete) method declared in an interface may be overridden to be abstract in a derived interface

```
interface IA
{
    void M() { WriteLine("IA.M"); }
}
interface IB : IA
{
    abstract void IA.M();
}
class C : IB { } // error: class 'C' does not implement 'IA.M'.
```

The `abstract` modifier is not required in the declaration of `IB.M` (that is the default in interfaces), but it is probably good practice to be explicit in an override declaration.

This is useful in derived interfaces where the default implementation of a method is inappropriate and a more appropriate implementation should be provided by implementing classes.

Open Issue: Should reabstraction be permitted?

The most specific override rule

We require that every interface and class have a *most specific override* for every virtual member among the overrides appearing in the type or its direct and indirect interfaces. The *most specific override* is a unique override that is more specific than every other override. If there is no override, the member itself is considered the most specific override.

One override `M1` is considered *more specific* than another override `M2` if `M1` is declared on type `T1`, `M2` is declared on type `T2`, and either

1. `T1` contains `T2` among its direct or indirect interfaces, or
2. `T2` is an interface type but `T1` is not an interface type.

For example:

```
interface IA
{
    void M() { WriteLine("IA.M"); }
}
interface IB : IA
{
    void IA.M() { WriteLine("IB.M"); }
}
interface IC : IA
{
    void IA.M() { WriteLine("IC.M"); }
}
interface ID : IB, IC { } // error: no most specific override for 'IA.M'
abstract class C : IB, IC { } // error: no most specific override for 'IA.M'
abstract class D : IA, IB, IC // ok
{
    public abstract void M();
}
```

The most specific override rule ensures that a conflict (i.e. an ambiguity arising from diamond inheritance) is resolved explicitly by the programmer at the point where the conflict arises.

Because we support explicit abstract overrides in interfaces, we could do so in classes as well

```
abstract class E : IA, IB, IC // ok
{
    abstract void IA.M();
}
```

Open issue: should we support explicit interface abstract overrides in classes?

In addition, it is an error if in a class declaration the most specific override of some interface method is an abstract override that was declared in an interface. This is an existing rule restated using the new terminology.

```
interface IF
{
    void M();
}
abstract class F : IF { } // error: 'F' does not implement 'IF.M'
```

It is possible for a virtual property declared in an interface to have a most specific override for its `get` accessor in one interface and a most specific override for its `set` accessor in a different interface. This is considered a violation of the *most specific override* rule.

`static` and `private` methods

Because interfaces may now contain executable code, it is useful to abstract common code into private and static methods. We now permit these in interfaces.

Closed issue: Should we support private methods? Should we support static methods? **Decision: YES**

Open issue: should we permit interface methods to be `protected` or `internal` or other access? If so, what are the semantics? Are they `virtual` by default? If so, is there a way to make them non-virtual?

Open issue: If we support static methods, should we support (static) operators?

Base interface invocations

Code in a type that derives from an interface with a default method can explicitly invoke that interface's "base" implementation.

```
interface I0
{
    void M() { Console.WriteLine("I0"); }
}
interface I1 : I0
{
    override void M() { Console.WriteLine("I1"); }
}
interface I2 : I0
{
    override void M() { Console.WriteLine("I2"); }
}
interface I3 : I1, I2
{
    // an explicit override that invoke's a base interface's default method
    void I0.M() { I2.base.M(); }
}
```

An instance (nonstatic) method is permitted to invoke the implementation of an accessible instance method in a direct base interface nonvirtually by naming it using the syntax `base(Type).M`. This is useful when an override that is required to be provided due to diamond inheritance is resolved by delegating to one particular base implementation.


```

interface IA
{
    void M() { WriteLine("IA.M"); }
}
interface IB : IA
{
    override void IA.M() { WriteLine("IB.M"); }
}
interface IC : IA
{
    override void IA.M() { WriteLine("IC.M"); }
}

class D : IA, IB, IC
{
    void IA.M() { base(IB).M(); }
}

```

When a `virtual` or `abstract` member is accessed using the syntax `base(Type).M`, it is required that `Type` contains a unique *most specific override* for `M`.

Binding base clauses

Interfaces now contain types. These types may be used in the base clause as base interfaces. When binding a base clause, we may need to know the set of base interfaces to bind those types (e.g. to lookup in them and to resolve protected access). The meaning of an interface's base clause is thus circularly defined. To break the cycle, we add a new language rules corresponding to a similar rule already in place for classes.

While determining the meaning of the *interface_base* of an interface, the base interfaces are temporarily assumed to be empty. Intuitively this ensures that the meaning of a base clause cannot recursively depend on itself.

We used to have the following rules:

"When a class B derives from a class A, it is a compile-time error for A to depend on B. A class **directly depends on** its direct base class (if any) and **directly depends on** the ~~class~~ within which it is immediately nested (if any). Given this definition, the complete set of ~~classes~~ upon which a class depends is the reflexive and transitive closure of the **directly depends on** relationship."

It is a compile-time error for an interface to directly or indirectly inherit from itself. The **base interfaces** of an interface are the explicit base interfaces and their base interfaces. In other words, the set of base interfaces is the complete transitive closure of the explicit base interfaces, their explicit base interfaces, and so on.

We are adjusting them as follows:

When a class B derives from a class A, it is a compile-time error for A to depend on B. A class **directly depends on** its direct base class (if any) and **directly depends on** the *type* within which it is immediately nested (if any).

When an interface IB extends an interface IA, it is a compile-time error for IA to depend on IB. An interface **directly depends on** its direct base interfaces (if any) and **directly depends on** the type within which it is immediately nested (if any).

Given these definitions, the complete set of **types** upon which a type depends is the reflexive and transitive closure of the **directly depends on** relationship.

Effect on existing programs

The rules presented here are intended to have no effect on the meaning of existing programs.

Example 1:

```

interface IA
{
    void M();
}
class C: IA // Error: IA.M has no concrete most specific override in C
{
    public static void M() { } // method unrelated to 'IA.M' because static
}

```

Example 2:

```

interface IA
{
    void M();
}
class Base: IA
{
    void IA.M() { }
}
class Derived: Base, IA // OK, all interface members have a concrete most specific override
{
    private void M() { } // method unrelated to 'IA.M' because private
}

```

The same rules give similar results to the analogous situation involving default interface methods:

```

interface IA
{
    void M() { }
}
class Derived: IA // OK, all interface members have a concrete most specific override
{
    private void M() { } // method unrelated to 'IA.M' because private
}

```

Closed issue: confirm that this is an intended consequence of the specification. **Decision: YES**

Runtime method resolution

Closed Issue: The spec should describe the runtime method resolution algorithm in the face of interface default methods. We need to ensure that the semantics are consistent with the language semantics, e.g. which declared methods do and do not override or implement an `internal` method.

CLR support API

In order for compilers to detect when they are compiling for a runtime that supports this feature, libraries for such runtimes are modified to advertise that fact through the API discussed in

<https://github.com/dotnet/corefx/issues/17116>. We add

```

namespace System.Runtime.CompilerServices
{
    public static class RuntimeFeature
    {
        // Presence of the field indicates runtime support
        public const string DefaultInterfaceImplementation = nameof(DefaultInterfaceImplementation);
    }
}

```

Open issue: Is that the best name for the *CLR* feature? The CLR feature does much more than just that (e.g. relaxes protection constraints, supports overrides in interfaces, etc). Perhaps it should be called something like "concrete methods in interfaces", or "traits"?

Further areas to be specified

- [] It would be useful to catalog the kinds of source and binary compatibility effects caused by adding default interface methods and overrides to existing interfaces.

Drawbacks

This proposal requires a coordinated update to the CLR specification (to support concrete methods in interfaces and method resolution). It is therefore fairly "expensive" and it may be worth doing in combination with other features that we also anticipate would require CLR changes.

Alternatives

None.

Unresolved questions

- Open questions are called out throughout the proposal, above.
- See also <https://github.com/dotnet/csharplang/issues/406> for a list of open questions.
- The detailed specification must describe the resolution mechanism used at runtime to select the precise method to be invoked.
- The interaction of metadata produced by new compilers and consumed by older compilers needs to be worked out in detail. For example, we need to ensure that the metadata representation that we use does not cause the addition of a default implementation in an interface to break an existing class that implements that interface when compiled by an older compiler. This may affect the metadata representation that we can use.
- The design must consider interoperability with other languages and existing compilers for other languages.

Resolved Questions

Abstract Override

The earlier draft spec contained the ability to "reabstract" an inherited method:

```
interface IA
{
    void M();
}
interface IB : IA
{
    override void M() { }
}
interface IC : IB
{
    override void M(); // make it abstract again
}
```

My notes for 2017-03-20 showed that we decided not to allow this. However, there are at least two use cases for it:

1. The Java APIs, with which some users of this feature hope to interoperate, depend on this facility.
2. Programming with *traits* benefits from this. Reabstraction is one of the elements of the "traits" language feature ([https://en.wikipedia.org/wiki/Trait_\(computer_programming\)](https://en.wikipedia.org/wiki/Trait_(computer_programming))). The following is permitted with

classes:

```
public abstract class Base
{
    public abstract void M();
}
public abstract class A : Base
{
    public override void M() { }
}
public abstract class B : A
{
    public override abstract void M(); // reabstract Base.M
}
```

Unfortunately this code cannot be refactored as a set of interfaces (traits) unless this is permitted. By the *Jared principle of greed*, it should be permitted.

Closed issue: Should reabstraction be permitted? [YES] My notes were wrong. The [LDM notes](#) say that reabstraction is permitted in an interface. Not in a class.

Virtual Modifier vs Sealed Modifier

From [Aleksey Tsingauz](#):

We decided to allow modifiers explicitly stated on interface members, unless there is a reason to disallow some of them. This brings an interesting question around virtual modifier. Should it be required on members with default implementation?

We could say that:

- if there is no implementation and neither virtual, nor sealed are specified, we assume the member is abstract.
- if there is an implementation and neither abstract, nor sealed are specified, we assume the member is virtual.
- sealed modifier is required to make a method neither virtual, nor abstract.

Alternatively, we could say that virtual modifier is required for a virtual member. I.e, if there is a member with implementation not explicitly marked with virtual modifier, it is neither virtual, nor abstract. This approach might provide better experience when a method is moved from a class to an interface:

- an abstract method stays abstract.
- a virtual method stays virtual.
- a method without any modifier stays neither virtual, nor abstract.
- sealed modifier cannot be applied to a method that is not an override.

What do you think?

Closed Issue: Should a concrete method (with implementation) be implicitly `virtual`? [YES]

Decisions: Made in the LDM 2017-04-05:

1. non-virtual should be explicitly expressed through `sealed` or `private`.
2. `sealed` is the keyword to make interface instance members with bodies non-virtual
3. We want to allow all modifiers in interfaces
4. Default accessibility for interface members is public, including nested types

- private function members in interfaces are implicitly sealed, and `sealed` is not permitted on them.
- Private classes (in interfaces) are permitted and can be sealed, and that means sealed in the class sense of sealed.
- Absent a good proposal, partial is still not allowed on interfaces or their members.

Binary Compatibility 1

When a library provides a default implementation

```
interface I1
{
    void M() { Impl1 }
}
interface I2 : I1
{
}
class C : I2
{
}
```

We understand that the implementation of `I1.M` in `C` is `I1.M`. What if the assembly containing `I2` is changed as follows and recompiled

```
interface I2 : I1
{
    override void M() { Impl2 }
}
```

but `C` is not recompiled. What happens when the program is run? An invocation of `(C as I1).M()`

- Runs `I1.M`
- Runs `I2.M`
- Throws some kind of runtime error

Decision: Made 2017-04-11: Runs `I2.M`, which is the unambiguously most specific override at runtime.

Event accessors (closed)

Closed Issue: Can an event be overridden "piecewise"?

Consider this case:

```
public interface I1
{
    event T e1;
}
public interface I2 : I1
{
    override event T
    {
        add { }
        // error: "remove" accessor missing
    }
}
```

This "partial" implementation of the event is not permitted because, as in a class, the syntax for an event declaration does not permit only one accessor; both (or neither) must be provided. You could accomplish the same thing by permitting the abstract remove accessor in the syntax to be implicitly abstract by the absence of a

body:

```
public interface I1
{
    event T e1;
}
public interface I2 : I1
{
    override event T
    {
        add { }
        remove; // implicitly abstract
    }
}
```

Note that *this is a new (proposed) syntax*. In the current grammar, event accessors have a mandatory body.

Closed Issue: Can an event accessor be (implicitly) abstract by the omission of a body, similarly to the way that methods in interfaces and property accessors are (implicitly) abstract by the omission of a body?

Decision: (2017-04-18) No, event declarations require both concrete accessors (or neither).

Reabstraction in a Class (closed)

Closed Issue: We should confirm that this is permitted (otherwise adding a default implementation would be a breaking change):

```
interface I1
{
    void M() { }
}
abstract class C : I1
{
    public abstract void M(); // implement I1.M with an abstract method in C
}
```

Decision: (2017-04-18) Yes, adding a body to an interface member declaration shouldn't break C.

Sealed Override (closed)

The previous question implicitly assumes that the `sealed` modifier can be applied to an `override` in an interface. This contradicts the draft specification. Do we want to permit sealing an override? Source and binary compatibility effects of sealing should be considered.

Closed Issue: Should we permit sealing an override?

Decision: (2017-04-18) Let's not allow `sealed` on overrides in interfaces. The only use of `sealed` on interface members is to make them non-virtual in their initial declaration.

Diamond inheritance and classes (closed)

The draft of the proposal prefers class overrides to interface overrides in diamond inheritance scenarios:

We require that every interface and class have a *most specific override* for every interface method among the overrides appearing in the type or its direct and indirect interfaces. The *most specific override* is a unique override that is more specific than every other override. If there is no override, the method itself is considered the most specific override.

One override `M1` is considered *more specific* than another override `M2` if `M1` is declared on type `T1`, `M2` is declared on type `T2`, and either

1. `T1` contains `T2` among its direct or indirect interfaces, or
2. `T2` is an interface type but `T1` is not an interface type.

The scenario is this

```
interface IA
{
    void M();
}
interface IB : IA
{
    override void M() { WriteLine("IB"); }
}
class Base : IA
{
    void IA.M() { WriteLine("Base"); }
}
class Derived : Base, IB // allowed?
{
    static void Main()
    {
        Ia a = new Derived();
        a.M();           // what does it do?
    }
}
```

We should confirm this behavior (or decide otherwise)

Closed Issue: Confirm the draft spec, above, for *most specific override* as it applies to mixed classes and interfaces (a class takes priority over an interface). See <https://github.com/dotnet/csharplang/blob/master/meetings/2017/LDM-2017-04-19.md#diamonds-with-classes>.

Interface methods vs structs (closed)

There are some unfortunate interactions between default interface methods and structs.

```
interface IA
{
    public void M() { }
}
struct S : IA
{
}
```

Note that interface members are not inherited:

```
var s = default(S);
s.M(); // error: 'S' does not contain a member 'M'
```

Consequently, the client must box the struct to invoke interface methods

```
IA s = default(S); // an S, boxed
s.M(); // ok
```

Boxing in this way defeats the principal benefits of a `struct` type. Moreover, any mutation methods will have no apparent effect, because they are operating on a *boxed copy* of the struct:

```

interface IB
{
    public void Increment() { P += 1; }
    public int P { get; set; }
}
struct T : IB
{
    public int P { get; set; } // auto-property
}

T t = default(T);
Console.WriteLine(t.P); // prints 0
(t as IB).Increment();
Console.WriteLine(t.P); // prints 0

```

Closed Issue: What can we do about this:

1. Forbid a `struct` from inheriting a default implementation. All interface methods would be treated as abstract in a `struct`. Then we may take time later to decide how to make it work better.
2. Come up with some kind of code generation strategy that avoids boxing. Inside a method like `IB.Increment`, the type of `this` would perhaps be akin to a type parameter constrained to `IB`. In conjunction with that, to avoid boxing in the caller, non-abstract methods would be inherited from interfaces. This may increase compiler and CLR implementation work substantially.
3. Not worry about it and just leave it as a wart.
4. Other ideas?

Decision: Not worry about it and just leave it as a wart. See

<https://github.com/dotnet/csharpplang/blob/master/meetings/2017/LDM-2017-04-19.md#structs-and-default-implementations>.

Base interface invocations (closed)

The draft spec suggests a syntax for base interface invocations inspired by Java: `Interface.base.M()`. We need to select a syntax, at least for the initial prototype. My favorite is `base<Interface>.M()`.

Closed Issue: What is the syntax for a base member invocation?

Decision: The syntax is `base(Interface).M()`. See

<https://github.com/dotnet/csharpplang/blob/master/meetings/2017/LDM-2017-04-19.md#base-invocation>. The interface so named must be a base interface, but does not need to be a direct base interface.

Open Issue: Should base interface invocations be permitted in class members?

Decision: Yes. <https://github.com/dotnet/csharpplang/blob/master/meetings/2017/LDM-2017-04-19.md#base-invocation>

Overriding non-public interface members (closed)

In an interface, non-public members from base interfaces are overridden using the `override` modifier. If it is an "explicit" override that names the interface containing the member, the access modifier is omitted.

Closed Issue: If it is an "implicit" override that does not name the interface, does the access modifier have to match?

Decision: Only public members may be implicitly overridden, and the access must match. See

<https://github.com/dotnet/csharpplang/blob/master/meetings/2017/LDM-2017-04-18.md#dim-implementing->

[a-non-public-interface-member-not-in-list.](#)

Open Issue: Is the access modifier required, optional, or omitted on an explicit override such as

```
override void IB.M() {} ?
```

Open Issue: Is `override` required, optional, or omitted on an explicit override such as `void IB.M() {}` ?

How does one implement a non-public interface member in a class? Perhaps it must be done explicitly?

```
interface IA
{
    internal void MI();
    protected void MP();
}
class C : IA
{
    // are these implementations?
    internal void MI() {}
    protected void MP() {}
}
```

Closed Issue: How does one implement a non-public interface member in a class?

Decision: You can only implement non-public interface members explicitly. See

<https://github.com/dotnet/csharplang/blob/master/meetings/2017/LDM-2017-04-18.md#dim-implementing-a-non-public-interface-member-not-in-list>.

Decision: No `override` keyword permitted on interface members.

<https://github.com/dotnet/csharplang/blob/master/meetings/2018/LDM-2018-10-17.md#does-an-override-in-an-interface-introduce-a-new-member>

Binary Compatibility 2 (closed)

Consider the following code in which each type is in a separate assembly

```
interface I1
{
    void M() { Impl1 }
}
interface I2 : I1
{
    override void M() { Impl2 }
}
interface I3 : I1
{
}
class C : I2, I3
{
}
```

We understand that the implementation of `I1.M` in `C` is `I2.M`. What if the assembly containing `I3` is changed as follows and recompiled

```
interface I3 : I1
{
    override void M() { Impl3 }
}
```

but `c` is not recompiled. What happens when the program is run? An invocation of `(C as I1).M()`

1. Runs `I1.M`
2. Runs `I2.M`
3. Runs `I3.M`
4. Either 2 or 3, deterministically
5. Throws some kind of runtime exception

Decision: Throw an exception (5). See

<https://github.com/dotnet/csharplang/blob/master/meetings/2018/LDM-2018-10-17.md#issues-in-default-interface-methods>.

Permit `partial` in interface? (closed)

Given that interfaces may be used in ways analogous to the way abstract classes are used, it may be useful to declare them `partial`. This would be particularly useful in the face of generators.

Proposal: Remove the language restriction that interfaces and members of interfaces may not be declared `partial`.

Decision: Yes. See <https://github.com/dotnet/csharplang/blob/master/meetings/2018/LDM-2018-10-17.md#permit-partial-in-interface>.

Main in an interface? (closed)

Open Issue: Is a `static Main` method in an interface a candidate to be the program's entry point?

Decision: Yes. See <https://github.com/dotnet/csharplang/blob/master/meetings/2018/LDM-2018-10-17.md#main-in-an-interface>.

Confirm intent to support public non-virtual methods (closed)

Can we please confirm (or reverse) our decision to permit non-virtual public methods in an interface?

```
interface IA
{
    public sealed void M() { }
}
```

Semi-Closed Issue: (2017-04-18) We think it is going to be useful, but will come back to it. This is a mental model tripping block.

Decision: Yes. <https://github.com/dotnet/csharplang/blob/master/meetings/2018/LDM-2018-10-17.md#confirm-that-we-support-public-non-virtual-methods>.

Does an `override` in an interface introduce a new member? (closed)

There are a few ways to observe whether an override declaration introduces a new member or not.

```

interface IA
{
    void M(int x) { }
}
interface IB : IA
{
    override void M(int y) { }
}
interface IC : IB
{
    static void M2()
    {
        M(y: 3); // permitted?
    }
    override void IB.M(int z) { } // permitted? What does it override?
}

```

Open Issue: Does an override declaration in an interface introduce a new member? (closed)

In a class, an overriding method is "visible" in some senses. For example, the names of its parameters take precedence over the names of parameters in the overridden method. It may be possible to duplicate that behavior in interfaces, as there is always a most specific override. But do we want to duplicate that behavior?

Also, it is possible to "override" an override method? [Moot]

Decision: No `override` keyword permitted on interface members.

<https://github.com/dotnet/csharpplang/blob/master/meetings/2018/LDM-2018-10-17.md#does-an-override-in-an-interface-introduce-a-new-member>.

Properties with a private accessor (closed)

We say that private members are not virtual, and the combination of virtual and private is disallowed. But what about a property with a private accessor?

```

interface IA
{
    public virtual int P
    {
        get => 3;
        private set => { }
    }
}

```

Is this allowed? Is the `set` accessor here `virtual` or not? Can it be overridden where it is accessible? Does the following implicitly implement only the `get` accessor?

```

class C : IA
{
    public int P
    {
        get => 4;
        set { }
    }
}

```

Is the following presumably an error because `IA.P.set` isn't virtual and also because it isn't accessible?

```

class C : IA
{
    int IA.P
    {
        get => 4;
        set { }
    }
}

```

Decision: The first example looks valid, while the last does not. This is resolved analogously to how it already works in C#. <https://github.com/dotnet/csharplang/blob/master/meetings/2018/LDM-2018-10-17.md#properties-with-a-private-accessor>

Base Interface Invocations, round 2 (closed)

Our previous "resolution" to how to handle base invocations doesn't actually provide sufficient expressiveness. It turns out that in C# and the CLR, unlike Java, you need to specify both the interface containing the method declaration and the location of the implementation you want to invoke.

I propose the following syntax for base calls in interfaces. I'm not in love with it, but it illustrates what any syntax must be able to express:

```

interface I1 { void M(); }
interface I2 { void M(); }
interface I3 : I1, I2 { void I1.M() { } void I2.M() { } }
interface I4 : I1, I2 { void I1.M() { } void I2.M() { } }
interface I5 : I3, I4
{
    void I1.M()
    {
        base<I3>(I1).M(); // calls I3's implementation of I1.M
        base<I4>(I1).M(); // calls I4's implementation of I1.M
    }
    void I2.M()
    {
        base<I3>(I2).M(); // calls I3's implementation of I2.M
        base<I4>(I2).M(); // calls I4's implementation of I2.M
    }
}

```

If there is no ambiguity, you can write it more simply

```

interface I1 { void M(); }
interface I3 : I1 { void I1.M() { } }
interface I4 : I1 { void I1.M() { } }
interface I5 : I3, I4
{
    void I1.M()
    {
        base<I3>.M(); // calls I3's implementation of I1.M
        base<I4>.M(); // calls I4's implementation of I1.M
    }
}

```

Or

```

interface I1 { void M(); }
interface I2 { void M(); }
interface I3 : I1, I2 { void I1.M() { } void I2.M() { } }
interface I5 : I3
{
    void I1.M()
    {
        base(I1).M(); // calls I3's implementation of I1.M
    }
    void I2.M()
    {
        base(I2).M(); // calls I3's implementation of I2.M
    }
}

```

Or

```

interface I1 { void M(); }
interface I3 : I1 { void I1.M() { } }
interface I5 : I3
{
    void I1.M()
    {
        base.M(); // calls I3's implementation of I1.M
    }
}

```

Decision: Decided on `base(N.I1<T>).M(s)`, conceding that if we have an invocation binding there may be problem here later on. <https://github.com/dotnet/csharpplang/blob/master/meetings/2018/LDM-2018-11-14.md#default-interface-implementations>

Warning for struct not implementing default method? (closed)

@vancem asserts that we should seriously consider producing a warning if a value type declaration fails to override some interface method, even if it would inherit an implementation of that method from an interface. Because it causes boxing and undermines constrained calls.

Decision: This seems like something more suited for an analyzer. It also seems like this warning could be noisy, since it would fire even if the default interface method is never called and no boxing will ever occur. <https://github.com/dotnet/csharpplang/blob/master/meetings/2018/LDM-2018-10-17.md#warning-for-struct-not-implementing-default-method>

Interface static constructors (closed)

When are interface static constructors run? The current CLI draft proposes that it occurs when the first static method or field is accessed. If there are neither of those then it might never be run??

[2018-10-09 The CLR team proposes "Going to mirror what we do for valuetypes (cctor check on access to each instance method)"]

Decision: Static constructors are also run on entry to instance methods, if the static constructor was not `beforefieldinit`, in which case static constructors are run before access to the first static field.

<https://github.com/dotnet/csharpplang/blob/master/meetings/2018/LDM-2018-10-17.md#when-are-interface-static-constructors-run>

Design meetings

[2017-03-08 LDM Meeting Notes](#) [2017-03-21 LDM Meeting Notes](#) [2017-03-23 meeting "CLR Behavior for Default Interface Methods"](#) [2017-04-05 LDM Meeting Notes](#) [2017-04-11 LDM Meeting Notes](#) [2017-04-18 LDM Meeting Notes](#) [2017-04-19 LDM Meeting Notes](#) [2017-05-17 LDM Meeting Notes](#) [2017-05-31 LDM Meeting](#)

Async Streams

12/28/2021 • 20 minutes to read • [Edit Online](#)

Summary

C# has support for iterator methods and async methods, but no support for a method that is both an iterator and an async method. We should rectify this by allowing for `await` to be used in a new form of `async` iterator, one that returns an `IAsyncEnumerable<T>` or `IAsyncEnumerator<T>` rather than an `IEnumerable<T>` or `IEnumerator<T>`, with `IAsyncEnumerable<T>` consumable in a new `await foreach`. An `IAsyncDisposable` interface is also used to enable asynchronous cleanup.

Related discussion

- <https://github.com/dotnet/roslyn/issues/261>
- <https://github.com/dotnet/roslyn/issues/114>

Detailed design

Interfaces

IAsyncDisposable

There has been much discussion of `IAsyncDisposable` (e.g. <https://github.com/dotnet/roslyn/issues/114>) and whether it's a good idea. However, it's a required concept to add in support of async iterators. Since `finally` blocks may contain `await`s, and since `finally` blocks need to be run as part of disposing of iterators, we need async disposal. It's also just generally useful any time cleaning up of resources might take any period of time, e.g. closing files (requiring flushes), deregistering callbacks and providing a way to know when deregistration has completed, etc.

The following interface is added to the core .NET libraries (e.g. System.Private.CoreLib / System.Runtime):

```
namespace System
{
    public interface IAsyncDisposable
    {
        ValueTask DisposeAsync();
    }
}
```

As with `Dispose`, invoking `DisposeAsync` multiple times is acceptable, and subsequent invocations after the first should be treated as nops, returning a synchronously completed successful task (`DisposeAsync` need not be thread-safe, though, and need not support concurrent invocation). Further, types may implement both `IDisposable` and `IAsyncDisposable`, and if they do, it's similarly acceptable to invoke `Dispose` and then `DisposeAsync` or vice versa, but only the first should be meaningful and subsequent invocations of either should be a nop. As such, if a type does implement both, consumers are encouraged to call once and only once the more relevant method based on the context, `Dispose` in synchronous contexts and `DisposeAsync` in asynchronous ones.

(I'm leaving discussion of how `IAsyncDisposable` interacts with `using` to a separate discussion. And coverage of how it interacts with `foreach` is handled later in this proposal.)

Alternatives considered:

- `DisposeAsync` *accepting a* `CancellationToken`: while in theory it makes sense that anything async can be canceled, disposal is about cleanup, closing things out, free'ing resources, etc., which is generally not something that should be canceled; cleanup is still important for work that's canceled. The same `CancellationToken` that caused the actual work to be canceled would typically be the same token passed to `DisposeAsync`, making `DisposeAsync` worthless because cancellation of the work would cause `DisposeAsync` to be a nop. If someone wants to avoid being blocked waiting for disposal, they can avoid waiting on the resulting `ValueTask`, or wait on it only for some period of time.
- `DisposeAsync` *returning a* `Task`: Now that a non-generic `ValueTask` exists and can be constructed from an `IValueTaskSource`, returning `ValueTask` from `DisposeAsync` allows an existing object to be reused as the promise representing the eventual async completion of `DisposeAsync`, saving a `Task` allocation in the case where `DisposeAsync` completes asynchronously.
- *Configuring* `DisposeAsync` *with a* `bool continueOnCapturedContext` (`ConfigureAwait`): While there may be issues related to how such a concept is exposed to `using`, `foreach`, and other language constructs that consume this, from an interface perspective it's not actually doing any `await` 'ing and there's nothing to configure... consumers of the `ValueTask` can consume it however they wish.
- `IAsyncDisposable` *inheriting* `IDisposable`: Since only one or the other should be used, it doesn't make sense to force types to implement both.
- `IDisposableAsync` *instead of* `IAsyncDisposable`: We've been following the naming that things/types are an "async something" whereas operations are "done async", so types have "Async" as a prefix and methods have "Async" as a suffix.

IAsyncEnumerable / IAsyncEnumerator

Two interfaces are added to the core .NET libraries:

```
namespace System.Collections.Generic
{
    public interface IAsyncEnumerable<out T>
    {
        IAsyncEnumerator<T> GetAsyncEnumerator(CancellationToken cancellationToken = default);
    }

    public interface IAsyncEnumerator<out T> : IAsyncDisposable
    {
        ValueTask<bool> MoveNextAsync();
        T Current { get; }
    }
}
```

Typical consumption (without additional language features) would look like:

```
IAsyncEnumerator<T> enumerator = enumerable.GetAsyncEnumerator();
try
{
    while (await enumerator.MoveNextAsync())
    {
        Use(enumerator.Current);
    }
}
finally { await enumerator.DisposeAsync(); }
```

Discarded options considered:

- `Task<bool> MoveNextAsync(); T current { get; }`: Using `Task<bool>` would support using a cached task object to represent synchronous, successful `MoveNextAsync` calls, but an allocation would still be required for

asynchronous completion. By returning `ValueTask<bool>`, we enable the enumerator object to itself implement `IValueTaskSource<bool>` and be used as the backing for the `ValueTask<bool>` returned from `MoveNextAsync`, which in turn allows for significantly reduced overheads.

- `ValueTask<(bool, T)> MoveNextAsync();` : It's not only harder to consume, but it means that `T` can no longer be covariant.
- `ValueTask<T?> TryMoveNextAsync();` : Not covariant.
- `Task<T?> TryMoveNextAsync();` : Not covariant, allocations on every call, etc.
- `ITask<T?> TryMoveNextAsync();` : Not covariant, allocations on every call, etc.
- `ITask<(bool, T)> TryMoveNextAsync();` : Not covariant, allocations on every call, etc.
- `Task<bool> TryMoveNextAsync(out T result);` : The `out` result would need to be set when the operation returns synchronously, not when it asynchronously completes the task potentially sometime long in the future, at which point there'd be no way to communicate the result.
- `IEnumerator<T>` *not implementing* `IDisposable` : We could choose to separate these. However, doing so complicates certain other areas of the proposal, as code must then be able to deal with the possibility that an enumerator doesn't provide disposal, which makes it difficult to write pattern-based helpers. Further, it will be common for enumerators to have a need for disposal (e.g. any C# async iterator that has a finally block, most things enumerating data from a network connection, etc.), and if one doesn't, it is simple to implement the method purely as `public ValueTask DisposeAsync() => default(ValueTask);` with minimal additional overhead.
- `IEnumerator<T> GetAsyncEnumerator();` : No cancellation token parameter.

Viable alternative:

```
namespace System.Collections.Generic
{
    public interface IAsyncEnumerable<out T>
    {
        IEnumerator<T> GetAsyncEnumerator();
    }

    public interface IAsyncEnumerator<out T> : IDisposable
    {
        ValueTask<bool> WaitForNextAsync();
        T TryGetNext(out bool success);
    }
}
```

`TryGetNext` is used in an inner loop to consume items with a single interface call as long as they're available synchronously. When the next item can't be retrieved synchronously, it returns false, and any time it returns false, a caller must subsequently invoke `WaitForNextAsync` to either wait for the next item to be available or to determine that there will never be another item. Typical consumption (without additional language features) would look like:

```

IEnumerator<T> enumerator = enumerable.GetAsyncEnumerator();
try
{
    while (await enumerator.WaitForNextAsync())
    {
        while (true)
        {
            int item = enumerator.TryGetNext(out bool success);
            if (!success) break;
            Use(item);
        }
    }
}
finally { await enumerator.DisposeAsync(); }

```

The advantage of this is two-fold, one minor and one major:

- *Minor: Allows for an enumerator to support multiple consumers.* There may be scenarios where it's valuable for an enumerator to support multiple concurrent consumers. That can't be achieved when `MoveNextAsync` and `Current` are separate such that an implementation can't make their usage atomic. In contrast, this approach provides a single method `TryGetNext` that supports pushing the enumerator forward and getting the next item, so the enumerator can enable atomicity if desired. However, it's likely that such scenarios could also be enabled by giving each consumer its own enumerator from a shared enumerable. Further, we don't want to enforce that every enumerator support concurrent usage, as that would add non-trivial overheads to the majority case that doesn't require it, which means a consumer of the interface generally couldn't rely on this any way.
- *Major: Performance.* The `MoveNextAsync` / `Current` approach requires two interface calls per operation, whereas the best case for `WaitForNextAsync` / `TryGetNext` is that most iterations complete synchronously, enabling a tight inner loop with `TryGetNext`, such that we only have one interface call per operation. This can have a measurable impact in situations where the interface calls dominate the computation.

However, there are non-trivial downsides, including significantly increased complexity when consuming these manually, and an increased chance of introducing bugs when using them. And while the performance benefits show up in microbenchmarks, we don't believe they'll be impactful in the vast majority of real usage. If it turns out they are, we can introduce a second set of interfaces in a light-up fashion.

Discarded options considered:

- `ValueTask<bool> WaitForNextAsync(); bool TryGetNext(out T result);` : `out` parameters can't be covariant. There's also a small impact here (an issue with the try pattern in general) that this likely incurs a runtime write barrier for reference type results.

Cancellation

There are several possible approaches to supporting cancellation:

1. `IAsyncEnumerable<T>` / `IAsyncEnumerator<T>` are cancellation-agnostic: `CancellationToken` doesn't appear anywhere. Cancellation is achieved by logically baking the `CancellationToken` into the enumerable and/or enumerator in whatever manner is appropriate, e.g. when calling an iterator, passing the `CancellationToken` as an argument to the iterator method and using it in the body of the iterator, as is done with any other parameter.
2. `IAsyncEnumerator<T>.GetAsyncEnumerator(CancellationToken)` : You pass a `CancellationToken` to `GetAsyncEnumerator`, and subsequent `MoveNextAsync` operations respect it however it can.
3. `IAsyncEnumerator<T>.MoveNextAsync(CancellationToken)` : You pass a `CancellationToken` to each individual `MoveNextAsync` call.
4. 1 && 2: You both embed `CancellationToken`s into your enumerable/enumerator and pass `CancellationToken`s into `GetAsyncEnumerator`.

5. 1 && 3: You both embed `CancellationToken`s into your enumerable/enumerator and pass `CancellationToken`s into `MoveNextAsync`.

From a purely theoretical perspective, (5) is the most robust, in that (a) `MoveNextAsync` accepting a `CancellationToken` enables the most fine-grained control over what's canceled, and (b) `CancellationToken` is just any other type that can be passed as an argument into iterators, embedded in arbitrary types, etc.

However, there are multiple problems with that approach:

- How does a `CancellationToken` passed to `GetAsyncEnumerator` make it into the body of the iterator? We could expose a new `iterator` keyword that you could dot off of to get access to the `CancellationToken` passed to `GetEnumerator`, but a) that's a lot of additional machinery, b) we're making it a very first-class citizen, and c) the 99% case would seem to be the same code both calling an iterator and calling `GetAsyncEnumerator` on it, in which case it can just pass the `CancellationToken` as an argument into the method.
- How does a `CancellationToken` passed to `MoveNextAsync` get into the body of the method? This is even worse, as if it's exposed off of an `iterator` local object, its value could change across awaits, which means any code that registered with the token would need to unregister from it prior to awaits and then re-register after; it's also potentially quite expensive to need to do such registering and unregistering in every `MoveNextAsync` call, regardless of whether implemented by the compiler in an iterator or by a developer manually.
- How does a developer cancel a `foreach` loop? If it's done by giving a `CancellationToken` to an enumerable/enumerator, then either a) we need to support `foreach` 'ing over enumerators, which raises them to being first-class citizens, and now you need to start thinking about an ecosystem built up around enumerators (e.g. LINQ methods) or b) we need to embed the `CancellationToken` in the enumerable anyway by having some `WithCancellation` extension method off of `IAsyncEnumerable<T>` that would store the provided token and then pass it into the wrapped enumerable's `GetAsyncEnumerator` when the `GetAsyncEnumerator` on the returned struct is invoked (ignoring that token). Or, you can just use the `CancellationToken` you have in the body of the `foreach`.
- If/when query comprehensions are supported, how would the `CancellationToken` supplied to `GetEnumerator` or `MoveNextAsync` be passed into each clause? The easiest way would simply be for the clause to capture it, at which point whatever token is passed to `GetAsyncEnumerator` / `MoveNextAsync` is ignored.

An earlier version of this document recommended (1), but we since switched to (4).

The two main problems with (1):

- producers of cancellable enumerables have to implement some boilerplate, and can only leverage the compiler's support for async-iterators to implement a `IAsyncEnumerable<T> GetAsyncEnumerator(CancellationToken)` method.
- it is likely that many producers would be tempted to just add a `CancellationToken` parameter to their async-enumerable signature instead, which will prevent consumers from passing the cancellation token they want when they are given an `IAsyncEnumerable` type.

There are two main consumption scenarios:

1. `await foreach (var i in GetData(token)) ...` where the consumer calls the async-iterator method,
2. `await foreach (var i in givenIAsyncEnumerable.WithCancellation(token)) ...` where the consumer deals with a given `IAsyncEnumerable` instance.

We find that a reasonable compromise to support both scenarios in a way that is convenient for both producers and consumers of async-streams is to use a specially annotated parameter in the async-iterator method. The `[EnumeratorCancellation]` attribute is used for this purpose. Placing this attribute on a parameter tells the compiler that if a token is passed to the `GetAsyncEnumerator` method, that token should be used instead of the

value originally passed for the parameter.

Consider `IAsyncEnumerable<int> GetData([EnumeratorCancellation] CancellationToken token = default)`. The implementer of this method can simply use the parameter in the method body. The consumer can use either consumption patterns above:

1. if you use `GetData(token)`, then the token is saved into the async-enumerable and will be used in iteration,
2. if you use `givenIAsyncEnumerable.WithCancellation(token)`, then the token passed to `GetAsyncEnumerator` will supersede any token saved in the async-enumerable.

foreach

`foreach` will be augmented to support `IAsyncEnumerable<T>` in addition to its existing support for `IEnumerable<T>`. And it will support the equivalent of `IAsyncEnumerable<T>` as a pattern if the relevant members are exposed publicly, falling back to using the interface directly if not, in order to enable struct-based extensions that avoid allocating as well as using alternative awaitables as the return type of `MoveNextAsync` and `DisposeAsync`.

Syntax

Using the syntax:

```
foreach (var i in enumerable)
```

C# will continue to treat `enumerable` as a synchronous enumerable, such that even if it exposes the relevant APIs for async enumerables (exposing the pattern or implementing the interface), it will only consider the synchronous APIs.

To force `foreach` to instead only consider the asynchronous APIs, `await` is inserted as follows:

```
await foreach (var i in enumerable)
```

No syntax would be provided that would support using either the async or the sync APIs; the developer must choose based on the syntax used.

Discarded options considered:

- `foreach (var i in await enumerable)`: This is already valid syntax, and changing its meaning would be a breaking change. This means to `await` the `enumerable`, get back something synchronously iterable from it, and then synchronously iterate through that.
- `foreach (var i await in enumerable)`, `foreach (var await i in enumerable)`, `foreach (await var i in enumerable)`: These all suggest that we're awaiting the next item, but there are other awaits involved in `foreach`, in particular if the enumerable is an `IAsyncDisposable`, we will be `await`'ing its async disposal. That `await` is as the scope of the `foreach` rather than for each individual element, and thus the `await` keyword deserves to be at the `foreach` level. Further, having it associated with the `foreach` gives us a way to describe the `foreach` with a different term, e.g. a "await foreach". But more importantly, there's value in considering `foreach` syntax at the same time as `using` syntax, so that they remain consistent with each other, and `using (await ...)` is already valid syntax.
- `foreach await (var i in enumerable)`

Still to consider:

- `foreach` today does not support iterating through an enumerator. We expect it will be more common to have `IAsyncEnumerator<T>`'s handed around, and thus it's tempting to support `await foreach` with both

`IEnumerable<T>` and `IAsyncEnumerator<T>`. But once we add such support, it introduces the question of whether `IAsyncEnumerator<T>` is a first-class citizen, and whether we need to have overloads of combinators that operate on enumerators in addition to enumerables? Do we want to encourage methods to return enumerators rather than enumerables? We should continue to discuss this. If we decide we don't want to support it, we might want to introduce an extension method

`public static IEnumerable<T> AsEnumerable<T>(this IAsyncEnumerator<T> enumerator);` that would allow an enumerator to still be `foreach`'d. If we decide we do want to support it, we'll need to also decide on whether the `await foreach` would be responsible for calling `DisposeAsync` on the enumerator, and the answer is likely "no, control over disposal should be handled by whoever called `GetEnumerator`."

Pattern-based Compilation

The compiler will bind to the pattern-based APIs if they exist, preferring those over using the interface (the pattern may be satisfied with instance methods or extension methods). The requirements for the pattern are:

- The enumerable must expose a `GetAsyncEnumerator` method that may be called with no arguments and that returns an enumerator that meets the relevant pattern.
- The enumerator must expose a `MoveNextAsync` method that may be called with no arguments and that returns something which may be `await`ed and whose `GetResult()` returns a `bool`.
- The enumerator must also expose `Current` property whose getter returns a `T` representing the kind of data being enumerated.
- The enumerator may optionally expose a `DisposeAsync` method that may be invoked with no arguments and that returns something that can be `await`ed and whose `GetResult()` returns `void`.

This code:

```
var enumerable = ...;
await foreach (T item in enumerable)
{
    ...
}
```

is translated to the equivalent of:

```
var enumerable = ...;
var enumerator = enumerable.GetAsyncEnumerator();
try
{
    while (await enumerator.MoveNextAsync())
    {
        T item = enumerator.Current;
        ...
    }
}
finally
{
    await enumerator.DisposeAsync(); // omitted, along with the try/finally, if the enumerator doesn't
    expose DisposeAsync
}
```

If the iterated type doesn't expose the right pattern, the interfaces will be used.

ConfigureAwait

This pattern-based compilation will allow `ConfigureAwait` to be used on all of the awaits, via a `ConfigureAwait` extension method:

```

await foreach (T item in enumerable.ConfigureAwait(false))
{
    ...
}

```

This will be based on types we'll add to .NET as well, likely to `System.Threading.Tasks.Extensions.dll`:

```

// Approximate implementation, omitting arg validation and the like
namespace System.Threading.Tasks
{
    public static class AsyncEnumerableExtensions
    {
        public static ConfiguredAsyncEnumerable<T> ConfigureAwait<T>(this IAsyncEnumerable<T> enumerable,
bool continueOnCapturedContext) =>
            new ConfiguredAsyncEnumerable<T>(enumerable, continueOnCapturedContext);

        public struct ConfiguredAsyncEnumerable<T>
        {
            private readonly IAsyncEnumerable<T> _enumerable;
            private readonly bool _continueOnCapturedContext;

            internal ConfiguredAsyncEnumerable(IAsyncEnumerable<T> enumerable, bool
continueOnCapturedContext)
            {
                _enumerable = enumerable;
                _continueOnCapturedContext = continueOnCapturedContext;
            }

            public ConfiguredAsyncEnumerator<T> GetAsyncEnumerator() =>
                new ConfiguredAsyncEnumerator<T>(_enumerable.GetAsyncEnumerator(),
_continueOnCapturedContext);

            public struct Enumerator
            {
                private readonly IAsyncEnumerator<T> _enumerator;
                private readonly bool _continueOnCapturedContext;

                internal Enumerator(IAsyncEnumerator<T> enumerator, bool continueOnCapturedContext)
                {
                    _enumerator = enumerator;
                    _continueOnCapturedContext = continueOnCapturedContext;
                }

                public ConfiguredValueTaskAwaitable<bool> MoveNextAsync() =>
                    _enumerator.MoveNextAsync().ConfigureAwait(_continueOnCapturedContext);

                public T Current => _enumerator.Current;

                public ConfiguredValueTaskAwaitable DisposeAsync() =>
                    _enumerator.DisposeAsync().ConfigureAwait(_continueOnCapturedContext);
            }
        }
    }
}

```

Note that this approach will not enable `ConfigureAwait` to be used with pattern-based enumerables, but then again it's already the case that the `ConfigureAwait` is only exposed as an extension on `Task` / `Task<T>` / `ValueTask` / `ValueTask<T>` and can't be applied to arbitrary awaitable things, as it only makes sense when applied to Tasks (it controls a behavior implemented in Task's continuation support), and thus doesn't make sense when using a pattern where the awaitable things may not be tasks. Anyone returning awaitable things can provide their own custom behavior in such advanced scenarios.

(If we can come up with some way to support a scope- or assembly-level `ConfigureAwait` solution, then this

won't be necessary.)

Async Iterators

The language / compiler will support producing `IAsyncEnumerable<T>` s and `IAsyncEnumerator<T>` s in addition to consuming them. Today the language supports writing an iterator like:

```
static IEnumerable<int> MyIterator()
{
    try
    {
        for (int i = 0; i < 100; i++)
        {
            Thread.Sleep(1000);
            yield return i;
        }
    }
    finally
    {
        Thread.Sleep(200);
        Console.WriteLine("finally");
    }
}
```

but `await` can't be used in the body of these iterators. We will add that support.

Syntax

The existing language support for iterators infers the iterator nature of the method based on whether it contains any `yield` s. The same will be true for async iterators. Such async iterators will be demarcated and differentiated from synchronous iterators via adding `async` to the signature, and must then also have either `IAsyncEnumerable<T>` or `IAsyncEnumerator<T>` as its return type. For example, the above example could be written as an async iterator as follows:

```
static async IAsyncEnumerable<int> MyIterator()
{
    try
    {
        for (int i = 0; i < 100; i++)
        {
            await Task.Delay(1000);
            yield return i;
        }
    }
    finally
    {
        await Task.Delay(200);
        Console.WriteLine("finally");
    }
}
```

Alternatives considered:

- *Not using `async` in the signature.* Using `async` is likely technically required by the compiler, as it uses it to determine whether `await` is valid in that context. But even if it's not required, we've established that `await` may only be used in methods marked as `async`, and it seems important to keep the consistency.
- *Enabling custom builders for `IAsyncEnumerable<T>`.* That's something we could look at for the future, but the machinery is complicated and we don't support that for the synchronous counterparts.
- *Having an `iterator` keyword in the signature.* Async iterators would use `async iterator` in the signature, and `yield` could only be used in `async` methods that included `iterator`; `iterator` would then be made

optional on synchronous iterators. Depending on your perspective, this has the benefit of making it very clear by the signature of the method whether `yield` is allowed and whether the method is actually meant to return instances of type `IAsyncEnumerable<T>` rather than the compiler manufacturing one based on whether the code uses `yield` or not. But it is different from synchronous iterators, which don't and can't be made to require one. Plus some developers don't like the extra syntax. If we were designing it from scratch, we'd probably make this required, but at this point there's much more value in keeping async iterators close to sync iterators.

LINQ

There are over ~200 overloads of methods on the `System.Linq.Enumerable` class, all of which work in terms of `IEnumerable<T>`; some of these accept `IEnumerable<T>`, some of them produce `IEnumerable<T>`, and many do both. Adding LINQ support for `IAsyncEnumerable<T>` would likely entail duplicating all of these overloads for it, for another ~200. And since `IAsyncEnumerator<T>` is likely to be more common as a standalone entity in the asynchronous world than `IEnumerator<T>` is in the synchronous world, we could potentially need another ~200 overloads that work with `IAsyncEnumerator<T>`. Plus, a large number of the overloads deal with predicates (e.g. `Where` that takes a `Func<T, bool>`), and it may be desirable to have `IAsyncEnumerable<T>`-based overloads that deal with both synchronous and asynchronous predicates (e.g. `Func<T, ValueTask<bool>>` in addition to `Func<T, bool>`). While this isn't applicable to all of the now ~400 new overloads, a rough calculation is that it'd be applicable to half, which means another ~200 overloads, for a total of ~600 new methods.

That is a staggering number of APIs, with the potential for even more when extension libraries like Interactive Extensions (Ix) are considered. But Ix already has an implementation of many of these, and there doesn't seem to be a great reason to duplicate that work; we should instead help the community improve Ix and recommend it for when developers want to use LINQ with `IAsyncEnumerable<T>`.

There is also the issue of query comprehension syntax. The pattern-based nature of query comprehensions would allow them to "just work" with some operators, e.g. if Ix provides the following methods:

```
public static IAsyncEnumerable<TResult> Select<TSource, TResult>(this IAsyncEnumerable<TSource> source,
    Func<TSource, TResult> func);
public static IAsyncEnumerable<T> Where(this IAsyncEnumerable<T> source, Func<T, bool> func);
```

then this C# code will "just work":

```
IAsyncEnumerable<int> enumerable = ...;
IAsyncEnumerable<int> result = from item in enumerable
    where item % 2 == 0
    select item * 2;
```

However, there is no query comprehension syntax that supports using `await` in the clauses, so if Ix added, for example:

```
public static IAsyncEnumerable<TResult> Select<TSource, TResult>(this IAsyncEnumerable<TSource> source,
    Func<TSource, ValueTask<TResult>> func);
```

then this would "just work":


```

IEnumerable<string> result = from url in urls
                             where item % 2 == 0
                             select SomeAsyncMethod(item);

async ValueTask<int> SomeAsyncMethod(int item)
{
    await Task.Yield();
    return item * 2;
}

```

but there'd be no way to write it with the `await` inline in the `select` clause. As a separate effort, we could look into adding `async { ... }` expressions to the language, at which point we could allow them to be used in query comprehensions and the above could instead be written as:

```

IEnumerable<int> result = from item in enumerable
                         where item % 2 == 0
                         select async
                        {
                            await Task.Yield();
                            return item * 2;
                        };

```

or to enabling `await` to be used directly in expressions, such as by supporting `async from`. However, it's unlikely a design here would impact the rest of the feature set one way or the other, and this isn't a particularly high-value thing to invest in right now, so the proposal is to do nothing additional here right now.

Integration with other asynchronous frameworks

Integration with `IObservable<T>` and other asynchronous frameworks (e.g. reactive streams) would be done at the library level rather than at the language level. For example, all of the data from an `IAsyncEnumerator<T>` can be published to an `IObserver<T>` simply by `await foreach`'ing over the enumerator and `OnNext`'ing the data to the observer, so an `AsObservable<T>` extension method is possible. Consuming an `IObservable<T>` in a `await foreach` requires buffering the data (in case another item is pushed while the previous item is still being processing), but such a push-pull adapter can easily be implemented to enable an `IObservable<T>` to be pulled from with an `IAsyncEnumerator<T>`. Etc. Rx/lx already provide prototypes of such implementations, and libraries like <https://github.com/dotnet/corefx/tree/master/src/System.Threading.Channels> provide various kinds of buffering data structures. The language need not be involved at this stage.

Ranges

12/28/2021 • 12 minutes to read • [Edit Online](#)

Summary

This feature is about delivering two new operators that allow constructing `System.Index` and `System.Range` objects, and using them to index/slice collections at runtime.

Overview

Well-known types and members

To use the new syntactic forms for `System.Index` and `System.Range`, new well-known types and members may be necessary, depending on which syntactic forms are used.

To use the "hat" operator (`^`), the following is required

```
namespace System
{
    public readonly struct Index
    {
        public Index(int value, bool fromEnd);
    }
}
```

To use the `System.Index` type as an argument in an array element access, the following member is required:

```
int System.Index.GetOffset(int length);
```

The `..` syntax for `System.Range` will require the `System.Range` type, as well as one or more of the following members:

```
namespace System
{
    public readonly struct Range
    {
        public Range(System.Index start, System.Index end);
        public static Range StartAt(System.Index start);
        public static Range EndAt(System.Index end);
        public static Range All { get; }
    }
}
```

The `..` syntax allows for either, both, or none of its arguments to be absent. Regardless of the number of arguments, the `Range` constructor is always sufficient for using the `Range` syntax. However, if any of the other members are present and one or more of the `..` arguments are missing, the appropriate member may be substituted.

Finally, for a value of type `System.Range` to be used in an array element access expression, the following member must be present:

```
namespace System.Runtime.CompilerServices
{
    public static class RuntimeHelpers
    {
        public static T[] GetSubArray<T>(T[] array, System.Range range);
    }
}
```

System.Index

C# has no way of indexing a collection from the end, but rather most indexers use the "from start" notion, or do a "length - i" expression. We introduce a new Index expression that means "from the end". The feature will introduce a new unary prefix "hat" operator. Its single operand must be convertible to `System.Int32`. It will be lowered into the appropriate `System.Index` factory method call.

We augment the grammar for *unary_expression* with the following additional syntax form:

```
unary_expression
: '^' unary_expression
;
```

We call this the *index from end* operator. The predefined *index from end* operators are as follows:

```
System.Index operator ^(int fromEnd);
```

The behavior of this operator is only defined for input values greater than or equal to zero.

Examples:

```
var array = new int[] { 1, 2, 3, 4, 5 };
var thirdItem = array[2];    // array[2]
var lastItem = array[^1];    // array[new Index(1, fromEnd: true)]
```

System.Range

C# has no syntactic way to access "ranges" or "slices" of collections. Usually users are forced to implement complex structures to filter/operate on slices of memory, or resort to LINQ methods like `list.Skip(5).Take(2)`. With the addition of `System.Span<T>` and other similar types, it becomes more important to have this kind of operation supported on a deeper level in the language/runtime, and have the interface unified.

The language will introduce a new range operator `x..y`. It is a binary infix operator that accepts two expressions. Either operand can be omitted (examples below), and they have to be convertible to `System.Index`. It will be lowered to the appropriate `System.Range` factory method call.

We replace the C# grammar rules for *multiplicative_expression* with the following (in order to introduce a new precedence level):

```

range_expression
: unary_expression
| range_expression? '..' range_expression?
;

multiplicative_expression
: range_expression
| multiplicative_expression '*' range_expression
| multiplicative_expression '/' range_expression
| multiplicative_expression '%' range_expression
;

```

All forms of the *range operator* have the same precedence. This new precedence group is lower than the *unary operators* and higher than the *multiplicative arithmetic operators*.

We call the `..` operator the *range operator*. The built-in range operator can roughly be understood to correspond to the invocation of a built-in operator of this form:

```
System.Range operator ..(Index start = 0, Index end = ^0);
```

Examples:

```

var array = new int[] { 1, 2, 3, 4, 5 };
var slice1 = array[2..^3];    // array[new Range(2, new Index(3, fromEnd: true))]
var slice2 = array[..^3];    // array[Range.EndAt(new Index(3, fromEnd: true))]
var slice3 = array[2..];     // array[Range.StartAt(2)]
var slice4 = array[..];      // array[Range.All]

```

Moreover, `System.Index` should have an implicit conversion from `System.Int32`, in order to avoid the need to overload mixing integers and indexes over multi-dimensional signatures.

Adding Index and Range support to existing library types

Implicit Index support

The language will provide an instance indexer member with a single parameter of type `Index` for types which meet the following criteria:

- The type is `Countable`.
- The type has an accessible instance indexer which takes a single `int` as the argument.
- The type does not have an accessible instance indexer which takes an `Index` as the first parameter. The `Index` must be the only parameter or the remaining parameters must be optional.

A type is ***Countable*** if it has a property named `Length` or `Count` with an accessible getter and a return type of `int`. The language can make use of this property to convert an expression of type `Index` into an `int` at the point of the expression without the need to use the type `Index` at all. In case both `Length` and `Count` are present, `Length` will be preferred. For simplicity going forward, the proposal will use the name `Length` to represent `Count` or `Length`.

For such types, the language will act as if there is an indexer member of the form `T this[Index index]` where `T` is the return type of the `int` based indexer including any `ref` style annotations. The new member will have the same `get` and `set` members with matching accessibility as the `int` indexer.

The new indexer will be implemented by converting the argument of type `Index` into an `int` and emitting a call to the `int` based indexer. For discussion purposes, let's use the example of `receiver[expr]`. The conversion of `expr` to `int` will occur as follows:

- When the argument is of the form `^expr2` and the type of `expr2` is `int`, it will be translated to `receiver.Length - expr2`.
- Otherwise, it will be translated as `expr.GetOffset(receiver.Length)`.

Regardless of the specific conversion strategy, the order of evaluation should be equivalent to the following:

1. `receiver` is evaluated;
2. `expr` is evaluated;
3. `length` is evaluated, if needed;
4. the `int` based indexer is invoked.

This allows for developers to use the `Index` feature on existing types without the need for modification. For example:

```
List<char> list = ...;
var value = list[^1];

// Gets translated to
var value = list[list.Count - 1];
```

The `receiver` and `Length` expressions will be spilled as appropriate to ensure any side effects are only executed once. For example:

```
class Collection {
    private int[] _array = new[] { 1, 2, 3 };

    public int Length {
        get {
            Console.WriteLine("Length ");
            return _array.Length;
        }
    }

    public int this[int index] => _array[index];
}

class SideEffect {
    Collection Get() {
        Console.WriteLine("Get ");
        return new Collection();
    }

    void Use() {
        int i = Get()[^1];
        Console.WriteLine(i);
    }
}
```

This code will print "Get Length 3".

Implicit Range support

The language will provide an instance indexer member with a single parameter of type `Range` for types which meet the following criteria:

- The type is Countable.
- The type has an accessible member named `Slice` which has two parameters of type `int`.
- The type does not have an instance indexer which takes a single `Range` as the first parameter. The `Range` must be the only parameter or the remaining parameters must be optional.

For such types, the language will bind as if there is an indexer member of the form `T this[Range range]` where `T` is the return type of the `Slice` method including any `ref` style annotations. The new member will also have matching accessibility with `Slice`.

When the `Range` based indexer is bound on an expression named `receiver`, it will be lowered by converting the `Range` expression into two values that are then passed to the `Slice` method. For discussion purposes, let's use the example of `receiver[expr]`.

The first argument of `Slice` will be obtained by converting the range typed expression in the following way:

- When `expr` is of the form `expr1..expr2` (where `expr2` can be omitted) and `expr1` has type `int`, then it will be emitted as `expr1`.
- When `expr` is of the form `^expr1..expr2` (where `expr2` can be omitted), then it will be emitted as `receiver.Length - expr1`.
- When `expr` is of the form `..expr2` (where `expr2` can be omitted), then it will be emitted as `0`.
- Otherwise, it will be emitted as `expr.Start.GetOffset(receiver.Length)`.

This value will be re-used in the calculation of the second `Slice` argument. When doing so it will be referred to as `start`. The second argument of `Slice` will be obtained by converting the range typed expression in the following way:

- When `expr` is of the form `expr1..expr2` (where `expr1` can be omitted) and `expr2` has type `int`, then it will be emitted as `expr2 - start`.
- When `expr` is of the form `expr1..^expr2` (where `expr1` can be omitted), then it will be emitted as `(receiver.Length - expr2) - start`.
- When `expr` is of the form `expr1..` (where `expr1` can be omitted), then it will be emitted as `receiver.Length - start`.
- Otherwise, it will be emitted as `expr.End.GetOffset(receiver.Length) - start`.

Regardless of the specific conversion strategy, the order of evaluation should be equivalent to the following:

1. `receiver` is evaluated;
2. `expr` is evaluated;
3. `length` is evaluated, if needed;
4. the `Slice` method is invoked.

The `receiver`, `expr`, and `length` expressions will be spilled as appropriate to ensure any side effects are only executed once. For example:

```

class Collection {
    private int[] _array = new[] { 1, 2, 3 };

    public int Length {
        get {
            Console.Write("Length ");
            return _array.Length;
        }
    }

    public int[] Slice(int start, int length) {
        var slice = new int[length];
        Array.Copy(_array, start, slice, 0, length);
        return slice;
    }
}

class SideEffect {
    Collection Get() {
        Console.Write("Get ");
        return new Collection();
    }

    void Use() {
        var array = Get()[0..2];
        Console.WriteLine(array.Length);
    }
}

```

This code will print "Get Length 2".

The language will special case the following known types:

- `string`: the method `Substring` will be used instead of `Slice`.
- `array`: the method `System.Runtime.CompilerServices.RuntimeHelpers.GetSubArray` will be used instead of `Slice`.

Alternatives

The new operators (`^` and `..`) are syntactic sugar. The functionality can be implemented by explicit calls to `System.Index` and `System.Range` factory methods, but it will result in a lot more boilerplate code, and the experience will be unintuitive.

IL Representation

These two operators will be lowered to regular indexer/method calls, with no change in subsequent compiler layers.

Runtime behavior

- Compiler can optimize indexers for built-in types like arrays and strings, and lower the indexing to the appropriate existing methods.
- `System.Index` will throw if constructed with a negative value.
- `^0` does not throw, but it translates to the length of the collection/enumerable it is supplied to.
- `Range.All` is semantically equivalent to `0..^0`, and can be deconstructed to these indices.

Considerations

Detect Indexable based on ICollection

The inspiration for this behavior was collection initializers. Using the structure of a type to convey that it had opted into a feature. In the case of collection initializers types can opt into the feature by implementing the interface `IEnumerable` (non generic).

This proposal initially required that types implement `ICollection` in order to qualify as Indexable. That required a number of special cases though:

- `ref struct`: these cannot implement interfaces yet types like `Span<T>` are ideal for index / range support.
- `string`: does not implement `ICollection` and adding that `interface` has a large cost.

This means to support key types special casing is already needed. The special casing of `string` is less interesting as the language does this in other areas (`foreach` lowering, constants, etc ...). The special casing of `ref struct` is more concerning as it's special casing an entire class of types. They get labeled as Indexable if they simply have a property named `Count` with a return type of `int`.

After consideration the design was normalized to say that any type which has a property `Count` / `Length` with a return type of `int` is Indexable. That removes all special casing, even for `string` and arrays.

Detect just Count

Detecting on the property names `Count` or `Length` does complicate the design a bit. Picking just one to standardize though is not sufficient as it ends up excluding a large number of types:

- Use `Length`: excludes pretty much every collection in `System.Collections` and sub-namespaces. Those tend to derive from `ICollection` and hence prefer `Count` over `length`.
- Use `Count`: excludes `string`, arrays, `Span<T>` and most `ref struct` based types

The extra complication on the initial detection of Indexable types is outweighed by its simplification in other aspects.

Choice of Slice as a name

The name `Slice` was chosen as it's the de-facto standard name for slice style operations in .NET. Starting with `netcoreapp2.1` all span style types use the name `Slice` for slicing operations. Prior to `netcoreapp2.1` there really aren't any examples of slicing to look to for an example. Types like `List<T>`, `ArraySegment<T>`, `SortedList<T>` would've been ideal for slicing but the concept didn't exist when types were added.

Thus, `Slice` being the sole example, it was chosen as the name.

Index target type conversion

Another way to view the `Index` transformation in an indexer expression is as a target type conversion. Instead of binding as if there is a member of the form `return_type this[Index]`, the language instead assigns a target typed conversion to `int`.

This concept could be generalized to all member access on Countable types. Whenever an expression with type `Index` is used as an argument to an instance member invocation and the receiver is Countable then the expression will have a target type conversion to `int`. The member invocations applicable for this conversion include methods, indexers, properties, extension methods, etc ... Only constructors are excluded as they have no receiver.

The target type conversion will be implemented as follows for any expression which has a type of `Index`. For discussion purposes lets use the example of `receiver[expr]`:

- When `expr` is of the form `^expr2` and the type of `expr2` is `int`, it will be translated to `receiver.Length - expr2`.
- Otherwise, it will be translated as `expr.GetOffset(receiver.Length)`.

The `receiver` and `Length` expressions will be spilled as appropriate to ensure any side effects are only executed once. For example:

```
class Collection {
    private int[] _array = new[] { 1, 2, 3 };

    public int Length {
        get {
            Console.Write("Length ");
            return _array.Length;
        }
    }

    public int GetAt(int index) => _array[index];
}

class SideEffect {
    Collection Get() {
        Console.Write("Get ");
        return new Collection();
    }

    void Use() {
        int i = Get().GetAt(^1);
        Console.WriteLine(i);
    }
}
```

This code will print "Get Length 3".

This feature would be beneficial to any member which had a parameter that represented an index. For example `List<T>.InsertAt`. This also has the potential for confusion as the language can't give any guidance as to whether or not an expression is meant for indexing. All it can do is convert any `Index` expression to `int` when invoking a member on a Countable type.

Restrictions:

- This conversion is only applicable when the expression with type `Index` is directly an argument to the member. It would not apply to any nested expressions.

Decisions made during implementation

- All members in the pattern must be instance members
- If a Length method is found but it has the wrong return type, continue looking for Count
- The indexer used for the Index pattern must have exactly one int parameter
- The Slice method used for the Range pattern must have exactly two int parameters
- When looking for the pattern members, we look for original definitions, not constructed members

Design meetings

- [Jan 10, 2018](#)
- [Jan 18, 2018](#)
- [Jan 22, 2018](#)
- [Dec 3, 2018](#)
- [Mar 25, 2019](#)
- [April 1st, 2019](#)
- [April 15, 2019](#)

"pattern-based using" and "using declarations"

12/28/2021 • 4 minutes to read • [Edit Online](#)

Summary

The language will add two new capabilities around the `using` statement in order to make resource management simpler: `using` should recognize a disposable pattern in addition to `IDisposable` and add a `using` declaration to the language.

Motivation

The `using` statement is an effective tool for resource management today but it requires quite a bit of ceremony. Methods that have a number of resources to manage can get syntactically bogged down with a series of `using` statements. This syntax burden is enough that most coding style guidelines explicitly have an exception around braces for this scenario.

The `using` declaration removes much of the ceremony here and gets C# on par with other languages that include resource management blocks. Additionally the pattern-based `using` lets developers expand the set of types that can participate here. In many cases removing the need to create wrapper types that only exist to allow for a values use in a `using` statement.

Together these features allow developers to simplify and expand the scenarios where `using` can be applied.

Detailed Design

using declaration

The language will allow for `using` to be added to a local variable declaration. Such a declaration will have the same effect as declaring the variable in a `using` statement at the same location.

```
if (...)
{
    using FileStream f = new FileStream(@"C:\users\jaredpar\using.md");
    // statements
}

// Equivalent to
if (...)
{
    using (FileStream f = new FileStream(@"C:\users\jaredpar\using.md"))
    {
        // statements
    }
}
```

The lifetime of a `using` local will extend to the end of the scope in which it is declared. The `using` locals will then be disposed in the reverse order in which they are declared.

```

{
    using var f1 = new FileStream("...");
    using var f2 = new FileStream("..."), f3 = new FileStream("...");
    ...
    // Dispose f3
    // Dispose f2
    // Dispose f1
}

```

There are no restrictions around `goto`, or any other control flow construct in the face of a `using` declaration. Instead the code acts just as it would for the equivalent `using` statement:

```

{
    using var f1 = new FileStream("...");
target:
    using var f2 = new FileStream("...");
    if (someCondition)
    {
        // Causes f2 to be disposed but has no effect on f1
        goto target;
    }
}

```

A local declared in a `using` local declaration will be implicitly read-only. This matches the behavior of locals declared in a `using` statement.

The language grammar for `using` declarations will be the following:

```

local-using-declaration:
    using type using-declarators

using-declarators:
    using-declarator
    using-declarators , using-declarator

using-declarator:
    identifier = expression

```

Restrictions around `using` declaration:

- May not appear directly inside a `case` label but instead must be within a block inside the `case` label.
- May not appear as part of an `out` variable declaration.
- Must have an initializer for each declarator.
- The local type must be implicitly convertible to `IDisposable` or fulfill the `using` pattern.

pattern-based using

The language will add the notion of a disposable pattern: that is a type which has an accessible `Dispose` instance method. Types which fit the disposable pattern can participate in a `using` statement or declaration without being required to implement `IDisposable`.

```

class Resource
{
    public void Dispose() { ... }
}

using (var r = new Resource())
{
    // statements
}

```

This will allow developers to leverage `using` in a number of new scenarios:

- `ref struct`: These types can't implement interfaces today and hence can't participate in `using` statements.
- Extension methods will allow developers to augment types in other assemblies to participate in `using` statements.

In the situation where a type can be implicitly converted to `IDisposable` and also fits the disposable pattern, then `IDisposable` will be preferred. While this takes the opposite approach of `foreach` (pattern preferred over interface) it is necessary for backwards compatibility.

The same restrictions from a traditional `using` statement apply here as well: local variables declared in the `using` are read-only, a `null` value will not cause an exception to be thrown, etc ... The code generation will be different only in that there will not be a cast to `IDisposable` before calling `Dispose`:

```

{
    Resource r = new Resource();
    try {
        // statements
    }
    finally {
        if (r != null) r.Dispose();
    }
}

```

In order to fit the disposable pattern the `Dispose` method must be accessible, parameterless and have a `void` return type. There are no other restrictions. This explicitly means that extension methods can be used here.

Considerations

case labels without blocks

A `using declaration` is illegal directly inside a `case` label due to complications around its actual lifetime. One potential solution is to simply give it the same lifetime as an `out var` in the same location. It was deemed the extra complexity to the feature implementation and the ease of the work around (just add a block to the `case` label) didn't justify taking this route.

Future Expansions

fixed locals

A `fixed` statement has all of the properties of `using` statements that motivated the ability to have `using` locals. Consideration should be given to extending this feature to `fixed` locals as well. The lifetime and ordering rules should apply equally well for `using` and `fixed` here.

Static local functions

12/28/2021 • 2 minutes to read • [Edit Online](#)

Summary

Support local functions that disallow capturing state from the enclosing scope.

Motivation

Avoid unintentionally capturing state from the enclosing context. Allow local functions to be used in scenarios where a `static` method is required.

Detailed design

A local function declared `static` cannot capture state from the enclosing scope. As a result, locals, parameters, and `this` from the enclosing scope are not available within a `static` local function.

A `static` local function cannot reference instance members from an implicit or explicit `this` or `base` reference.

A `static` local function may reference `static` members from the enclosing scope.

A `static` local function may reference `constant` definitions from the enclosing scope.

`nameof()` in a `static` local function may reference locals, parameters, or `this` or `base` from the enclosing scope.

Accessibility rules for `private` members in the enclosing scope are the same for `static` and non-`static` local functions.

A `static` local function definition is emitted as a `static` method in metadata, even if only used in a delegate.

A non-`static` local function or lambda can capture state from an enclosing `static` local function but cannot capture state outside the enclosing `static` local function.

A `static` local function cannot be invoked in an expression tree.

A call to a local function is emitted as `call` rather than `callvirt`, regardless of whether the local function is `static`.

Overload resolution of a call within a local function not affected by whether the local function is `static`.

Removing the `static` modifier from a local function in a valid program does not change the meaning of the program.

Design meetings

<https://github.com/dotnet/csharpplang/blob/master/meetings/2018/LDM-2018-09-10.md#static-local-functions>

null coalescing assignment

12/28/2021 • 2 minutes to read • [Edit Online](#)

Summary

Simplifies a common coding pattern where a variable is assigned a value if it is null.

As part of this proposal, we will also loosen the type requirements on `??` to allow an expression whose type is an unconstrained type parameter to be used on the left-hand side.

Motivation

It is common to see code of the form

```
if (variable == null)
{
    variable = expression;
}
```

This proposal adds a non-overloadable binary operator to the language that performs this function.

There have been at least eight separate community requests for this feature.

Detailed design

We add a new form of assignment operator

```
assignment_operator
: '??='
;
```

Which follows the [existing semantic rules for compound assignment operators](#), except that we elide the assignment if the left-hand side is non-null. The rules for this feature are as follows.

Given `a ??= b`, where `A` is the type of `a`, `B` is the type of `b`, and `A0` is the underlying type of `A` if `A` is a nullable value type:

1. If `A` does not exist or is a non-nullable value type, a compile-time error occurs.
2. If `B` is not implicitly convertible to `A` or `A0` (if `A0` exists), a compile-time error occurs.
3. If `A0` exists and `B` is implicitly convertible to `A0`, and `B` is not dynamic, then the type of `a ??= b` is `A0`.

`a ??= b` is evaluated at runtime as:

```
var tmp = a.GetValueOrDefault();
if (!a.HasValue) { tmp = b; a = tmp; }
tmp
```

Except that `a` is only evaluated once.

4. Otherwise, the type of `a ??= b` is `A`. `a ??= b` is evaluated at runtime as `a ?? (a = b)`, except that `a` is only evaluated once.

For the relaxation of the type requirements of `??`, we update the spec where it currently states that, given

`a ?? b`, where `A` is the type of `a`:

1. If `A` exists and is not a nullable type or a reference type, a compile-time error occurs.

We relax this requirement to:

1. If `A` exists and is a non-nullable value type, a compile-time error occurs.

This allows the null coalescing operator to work on unconstrained type parameters, as the unconstrained type parameter `T` exists, is not a nullable type, and is not a reference type.

Drawbacks

As with any language feature, we must question whether the additional complexity to the language is repaid in the additional clarity offered to the body of C# programs that would benefit from the feature.

Alternatives

The programmer can write `(x = x ?? y)`, `if (x == null) x = y;`, or `x ?? (x = y)` by hand.

Unresolved questions

- [] Requires LDM review
- [] Should we also support `&&=` and `||=` operators?

Design meetings

None.

Readonly Instance Members

12/28/2021 • 4 minutes to read • [Edit Online](#)

Championed Issue: <https://github.com/dotnet/csharplang/issues/1710>

Summary

Provide a way to specify individual instance members on a struct do not modify state, in the same way that `readonly struct` specifies no instance members modify state.

It is worth noting that `readonly instance member` \neq `pure instance member`. A `pure` instance member guarantees no state will be modified. A `readonly` instance member only guarantees that instance state will not be modified.

All instance members on a `readonly struct` could be considered implicitly `readonly instance members`. Explicit `readonly instance members` declared on non-readonly structs would behave in the same manner. For example, they would still create hidden copies if you called an instance member (on the current instance or on a field of the instance) which was itself not-readonly.

Motivation

Today, users have the ability to create `readonly struct` types which the compiler enforces that all fields are readonly (and by extension, that no instance members modify the state). However, there are some scenarios where you have an existing API that exposes accessible fields or that has a mix of mutating and non-mutating members. Under these circumstances, you cannot mark the type as `readonly` (it would be a breaking change).

This normally doesn't have much impact, except in the case of `in` parameters. With `in` parameters for non-readonly structs, the compiler will make a copy of the parameter for each instance member invocation, since it cannot guarantee that the invocation does not modify internal state. This can lead to a multitude of copies and worse overall performance than if you had just passed the struct directly by value. For an example, see this code on [sharplab](#)

Some other scenarios where hidden copies can occur include `static readonly fields` and `literals`. If they are supported in the future, `blittable constants` would end up in the same boat; that is they all currently necessitate a full copy (on instance member invocation) if the struct is not marked `readonly`.

Design

Allow a user to specify that an instance member is, itself, `readonly` and does not modify the state of the instance (with all the appropriate verification done by the compiler, of course). For example:


```

public struct Vector2
{
    public float x;
    public float y;

    public readonly float GetLengthReadOnly()
    {
        return MathF.Sqrt(LengthSquared);
    }

    public float GetLength()
    {
        return MathF.Sqrt(LengthSquared);
    }

    public readonly float GetLengthIllegal()
    {
        var tmp = MathF.Sqrt(LengthSquared);

        x = tmp;    // Compiler error, cannot write x
        y = tmp;    // Compiler error, cannot write y

        return tmp;
    }

    public readonly float LengthSquared
    {
        get
        {
            return (x * x) +
                   (y * y);
        }
    }
}

public static class MyClass
{
    public static float ExistingBehavior(in Vector2 vector)
    {
        // This code causes a hidden copy, the compiler effectively emits:
        //     var tmpVector = vector;
        //     return tmpVector.GetLength();
        //
        // This is done because the compiler doesn't know that `GetLength()`
        // won't mutate `vector`.

        return vector.GetLength();
    }

    public static float ReadOnlyBehavior(in Vector2 vector)
    {
        // This code is emitted exactly as listed. There are no hidden
        // copies as the `readonly` modifier indicates that the method
        // won't mutate `vector`.

        return vector.GetLengthReadOnly();
    }
}

```

ReadOnly can be applied to property accessors to indicate that `this` will not be mutated in the accessor. The following examples have readonly setters because those accessors modify the state of member field, but do not modify the value of that member field.

```
public readonly int Prop1
{
    get
    {
        return this._store["Prop1"];
    }
    set
    {
        this._store["Prop1"] = value;
    }
}
```

When `readonly` is applied to the property syntax, it means that all accessors are `readonly`.

```
public readonly int Prop2
{
    get
    {
        return this._store["Prop2"];
    }
    set
    {
        this._store["Prop2"] = value;
    }
}
```

Readonly can only be applied to accessors which do not mutate the containing type.

```
public int Prop3
{
    readonly get
    {
        return this._prop3;
    }
    set
    {
        this._prop3 = value;
    }
}
```

Readonly can be applied to some auto-implemented properties, but it won't have a meaningful effect. The compiler will treat all auto-implemented getters as readonly whether or not the `readonly` keyword is present.

```
// Allowed
public readonly int Prop4 { get; }
public int Prop5 { readonly get; }
public int Prop6 { readonly get; set; }

// Not allowed
public readonly int Prop7 { get; set; }
public int Prop8 { get; readonly set; }
```

Readonly can be applied to manually-implemented events, but not field-like events. Readonly cannot be applied to individual event accessors (add/remove).

```
// Allowed
public readonly event Action<EventArgs> Event1
{
    add { }
    remove { }
}

// Not allowed
public readonly event Action<EventArgs> Event2;
public event Action<EventArgs> Event3
{
    readonly add { }
    readonly remove { }
}
public static readonly event Event4
{
    add { }
    remove { }
}
```

Some other syntax examples:

- Expression bodied members: `public readonly float ExpressionBodiedMember => (x * x) + (y * y);`
- Generic constraints: `public readonly void GenericMethod<T>(T value) where T : struct { }`

The compiler would emit the instance member, as usual, and would additionally emit a compiler recognized attribute indicating that the instance member does not modify state. This effectively causes the hidden `this` parameter to become `in T` instead of `ref T`.

This would allow the user to safely call said instance method without the compiler needing to make a copy.

The restrictions would include:

- The `readonly` modifier cannot be applied to static methods, constructors or destructors.
- The `readonly` modifier cannot be applied to delegates.
- The `readonly` modifier cannot be applied to members of class or interface.

Drawbacks

Same drawbacks as exist with `readonly struct` methods today. Certain code may still cause hidden copies.

Notes

Using an attribute or another keyword may also be possible.

This proposal is somewhat related to (but is more a subset of) `functional purity` and/or `constant expressions`, both of which have had some existing proposals.

Permit `stackalloc` in nested contexts

12/28/2021 • 2 minutes to read • [Edit Online](#)

Stack allocation

We modify the section *Stack allocation* of the C# language specification to relax the places when a `stackalloc` expression may appear. We delete

```
local_variable_initializer_unsafe
    : stackalloc_initializer
    ;

stackalloc_initializer
    : 'stackalloc' unmanaged_type '[' expression ']'
    ;
```

and replace them with

```
primary_no_array_creation_expression
    : stackalloc_initializer
    ;

stackalloc_initializer
    : 'stackalloc' unmanaged_type '[' expression? ']' array_initializer?
    | 'stackalloc' '[' expression? ']' array_initializer
    ;
```

Note that the addition of an *array_initializer* to *stackalloc_initializer* (and making the index expression optional) was an [extension in C# 7.3](#) and is not described here.

The *element type* of the `stackalloc` expression is the *unmanaged_type* named in the *stackalloc* expression, if any, or the common type among the elements of the *array_initializer* otherwise.

The type of the *stackalloc_initializer* with *element type* `K` depends on its syntactic context:

- If the *stackalloc_initializer* appears directly as the *local_variable_initializer* of a *local_variable_declaration* statement or a *for_initializer*, then its type is `K*`.
- Otherwise its type is `System.Span<K>`.

Stackalloc Conversion

The *stackalloc conversion* is a new built-in implicit conversion from expression. When the type of a *stackalloc_initializer* is `K*`, there is an implicit *stackalloc conversion* from the *stackalloc_initializer* to the type `System.Span<K>`.

Records

12/28/2021 • 14 minutes to read • [Edit Online](#)

This proposal tracks the specification for the C# 9 records feature, as agreed to by the C# language design team.

The syntax for a record is as follows:

```
record_declaration
    : attributes? class_modifier* 'partial'? 'record' identifier type_parameter_list?
      parameter_list? record_base? type_parameter_constraints_clause* record_body
      ;

record_base
    : ':' class_type argument_list?
    | ':' interface_type_list
    | ':' class_type argument_list? ',' interface_type_list
    ;

record_body
    : '{' class_member_declaration* '}' ';'
    | ';'
    ;
```

Record types are reference types, similar to a class declaration. It is an error for a record to provide a `record_base` `argument_list` if the `record_declaration` does not contain a `parameter_list`. At most one partial type declaration of a partial record may provide a `parameter_list`.

Record parameters cannot use `ref`, `out` or `this` modifiers (but `in` and `params` are allowed).

Inheritance

Records cannot inherit from classes, unless the class is `object`, and classes cannot inherit from records. Records can inherit from other records.

Members of a record type

In addition to the members declared in the record body, a record type has additional synthesized members. Members are synthesized unless a member with a "matching" signature is declared in the record body or an accessible concrete non-virtual member with a "matching" signature is inherited. A matching member prevents the compiler from generating that member, not any other synthesized members. Two members are considered matching if they have the same signature or would be considered "hiding" in an inheritance scenario. It is an error for a member of a record to be named "Clone". It is an error for an instance field of a record to have an unsafe type.

The synthesized members are as follows:

Equality members

If the record is derived from `object`, the record type includes a synthesized readonly property equivalent to a property declared as follows:

```
Type EqualityContract { get; };
```

The property is `private` if the record type is `sealed`. Otherwise, the property is `virtual` and `protected`. The

property can be declared explicitly. It is an error if the explicit declaration does not match the expected signature or accessibility, or if the explicit declaration doesn't allow overriding it in a derived type and the record type is not `sealed`.

If the record type is derived from a base record type `Base`, the record type includes a synthesized readonly property equivalent to a property declared as follows:

```
protected override Type EqualityContract { get; };
```

The property can be declared explicitly. It is an error if the explicit declaration does not match the expected signature or accessibility, or if the explicit declaration doesn't allow overriding it in a derived type and the record type is not `sealed`. It is an error if either synthesized, or explicitly declared property doesn't override a property with this signature in the record type `Base` (for example, if the property is missing in the `Base`, or sealed, or not virtual, etc.). The synthesized property returns `typeof(R)` where `R` is the record type.

The record type implements `System.IEquatable<R>` and includes a synthesized strongly-typed overload of `Equals(R? other)` where `R` is the record type. The method is `public`, and the method is `virtual` unless the record type is `sealed`. The method can be declared explicitly. It is an error if the explicit declaration does not match the expected signature or accessibility, or the explicit declaration doesn't allow overriding it in a derived type and the record type is not `sealed`.

If `Equals(R? other)` is user-defined (not synthesized) but `GetHashCode` is not, a warning is produced.

```
public virtual bool Equals(R? other);
```

The synthesized `Equals(R?)` returns `true` if and only if each of the following are `true`:

- `other` is not `null`, and
- For each instance field `fieldN` in the record type that is not inherited, the value of `System.Collections.Generic.EqualityComparer<TN>.Default.Equals(fieldN, other.fieldN)` where `TN` is the field type, and
- If there is a base record type, the value of `base.Equals(other)` (a non-virtual call to `public virtual bool Equals(Base? other)`); otherwise the value of `EqualityContract == other.EqualityContract`.

The record type includes synthesized `==` and `!=` operators equivalent to operators declared as follows:

```
public static bool operator==(R? left, R? right)
    => (object)left == right || (left?.Equals(right) ?? false);
public static bool operator!=(R? left, R? right)
    => !(left == right);
```

The `Equals` method called by the `==` operator is the `Equals(R? other)` method specified above. The `!=` operator delegates to the `==` operator. It is an error if the operators are declared explicitly.

If the record type is derived from a base record type `Base`, the record type includes a synthesized override equivalent to a method declared as follows:

```
public sealed override bool Equals(Base? other);
```

It is an error if the override is declared explicitly. It is an error if the method doesn't override a method with same signature in record type `Base` (for example, if the method is missing in the `Base`, or sealed, or not virtual, etc.).

The synthesized override returns `Equals((object?)other)`.

The record type includes a synthesized override equivalent to a method declared as follows:

```
public override bool Equals(object? obj);
```

It is an error if the override is declared explicitly. It is an error if the method doesn't override `object.Equals(object? obj)` (for example, due to shadowing in intermediate base types, etc.). The synthesized override returns `Equals(other as R)` where `R` is the record type.

The record type includes a synthesized override equivalent to a method declared as follows:

```
public override int GetHashCode();
```

The method can be declared explicitly. It is an error if the explicit declaration doesn't allow overriding it in a derived type and the record type is not `sealed`. It is an error if either synthesized, or explicitly declared method doesn't override `object.GetHashCode()` (for example, due to shadowing in intermediate base types, etc.).

A warning is reported if one of `Equals(R?)` and `GetHashCode()` is explicitly declared but the other method is not explicit.

The synthesized override of `GetHashCode()` returns an `int` result of combining the following values:

- For each instance field `fieldN` in the record type that is not inherited, the value of `System.Collections.Generic.EqualityComparer<TN>.Default.GetHashCode(fieldN)` where `TN` is the field type, and
- If there is a base record type, the value of `base.GetHashCode()`; otherwise the value of `System.Collections.Generic.EqualityComparer<System.Type>.Default.GetHashCode(EqualityContract)`.

For example, consider the following record types:

```
record R1(T1 P1);  
record R2(T1 P1, T2 P2) : R1(P1);  
record R3(T1 P1, T2 P2, T3 P3) : R2(P1, P2);
```

For those record types, the synthesized equality members would be something like:

```

class R1 : IEquatable<R1>
{
    public T1 P1 { get; init; }
    protected virtual Type EqualityContract => typeof(R1);
    public override bool Equals(object? obj) => Equals(obj as R1);
    public virtual bool Equals(R1? other)
    {
        return !(other is null) &&
            EqualityContract == other.EqualityContract &&
            EqualityComparer<T1>.Default.Equals(P1, other.P1);
    }
    public static bool operator==(R1? left, R1? right)
        => (object)left == right || (left?.Equals(right) ?? false);
    public static bool operator!=(R1? left, R1? right)
        => !(left == right);
    public override int GetHashCode()
    {
        return Combine(EqualityComparer<Type>.Default.GetHashCode(EqualityContract),
            EqualityComparer<T1>.Default.GetHashCode(P1));
    }
}

class R2 : R1, IEquatable<R2>
{
    public T2 P2 { get; init; }
    protected override Type EqualityContract => typeof(R2);
    public override bool Equals(object? obj) => Equals(obj as R2);
    public sealed override bool Equals(R1? other) => Equals((object?)other);
    public virtual bool Equals(R2? other)
    {
        return base.Equals((R1?)other) &&
            EqualityComparer<T2>.Default.Equals(P2, other.P2);
    }
    public static bool operator==(R2? left, R2? right)
        => (object)left == right || (left?.Equals(right) ?? false);
    public static bool operator!=(R2? left, R2? right)
        => !(left == right);
    public override int GetHashCode()
    {
        return Combine(base.GetHashCode(),
            EqualityComparer<T2>.Default.GetHashCode(P2));
    }
}

class R3 : R2, IEquatable<R3>
{
    public T3 P3 { get; init; }
    protected override Type EqualityContract => typeof(R3);
    public override bool Equals(object? obj) => Equals(obj as R3);
    public sealed override bool Equals(R2? other) => Equals((object?)other);
    public virtual bool Equals(R3? other)
    {
        return base.Equals((R2?)other) &&
            EqualityComparer<T3>.Default.Equals(P3, other.P3);
    }
    public static bool operator==(R3? left, R3? right)
        => (object)left == right || (left?.Equals(right) ?? false);
    public static bool operator!=(R3? left, R3? right)
        => !(left == right);
    public override int GetHashCode()
    {
        return Combine(base.GetHashCode(),
            EqualityComparer<T3>.Default.GetHashCode(P3));
    }
}

```


Copy and Clone members

A record type contains two copying members:

- A constructor taking a single argument of the record type. It is referred to as a "copy constructor".
- A synthesized public parameterless instance "clone" method with a compiler-reserved name

The purpose of the copy constructor is to copy the state from the parameter to the new instance being created. This constructor doesn't run any instance field/property initializers present in the record declaration. If the constructor is not explicitly declared, a constructor will be synthesized by the compiler. If the record is sealed, the constructor will be private, otherwise it will be protected. An explicitly declared copy constructor must be either public or protected, unless the record is sealed. The first thing the constructor must do, is to call a copy constructor of the base, or a parameter-less object constructor if the record inherits from object. An error is reported if a user-defined copy constructor uses an implicit or explicit constructor initializer that doesn't fulfill this requirement. After a base copy constructor is invoked, a synthesized copy constructor copies values for all instance fields implicitly or explicitly declared within the record type. The sole presence of a copy constructor, whether explicit or implicit, doesn't prevent an automatic addition of a default instance constructor.

If a virtual "clone" method is present in the base record, the synthesized "clone" method overrides it and the return type of the method is the current containing type if the "covariant returns" feature is supported and the override return type otherwise. An error is produced if the base record clone method is sealed. If a virtual "clone" method is not present in the base record, the return type of the clone method is the containing type and the method is virtual, unless the record is sealed or abstract. If the containing record is abstract, the synthesized clone method is also abstract. If the "clone" method is not abstract, it returns the result of a call to a copy constructor.

Printing members: PrintMembers and ToString methods

If the record is derived from `object`, the record includes a synthesized method equivalent to a method declared as follows:

```
bool PrintMembers(System.Text.StringBuilder builder);
```

The method is `private` if the record type is `sealed`. Otherwise, the method is `virtual` and `protected`.

The method:

1. calls the method `System.Runtime.CompilerServices.RuntimeHelpers.EnsureSufficientExecutionStack()` if the method is present and the record has printable members.
2. for each of the record's printable members (non-static public field and readable property members), appends that member's name followed by " = " followed by the member's value separated with ";",
3. return true if the record has printable members.

For a member that has a value type, we will convert its value to a string representation using the most efficient method available to the target platform. At present that means calling `ToString` before passing to `StringBuilder.Append`.

If the record type is derived from a base record `Base`, the record includes a synthesized override equivalent to a method declared as follows:

```
protected override bool PrintMembers(StringBuilder builder);
```

If the record has no printable members, the method calls the base `PrintMembers` method with one argument (its `builder` parameter) and returns the result.

Otherwise, the method:

1. calls the base `PrintMembers` method with one argument (its `builder` parameter),
2. if the `PrintMembers` method returned true, append ", " to the builder,
3. for each of the record's printable members, appends that member's name followed by " = " followed by the member's value: `this.member` (or `this.member.ToString()` for value types), separated with ", ",
4. return true.

The `PrintMembers` method can be declared explicitly. It is an error if the explicit declaration does not match the expected signature or accessibility, or if the explicit declaration doesn't allow overriding it in a derived type and the record type is not `sealed`.

The record includes a synthesized method equivalent to a method declared as follows:

```
public override string ToString();
```

The method can be declared explicitly. It is an error if the explicit declaration does not match the expected signature or accessibility, or if the explicit declaration doesn't allow overriding it in a derived type and the record type is not `sealed`. It is an error if either synthesized, or explicitly declared method doesn't override `object.ToString()` (for example, due to shadowing in intermediate base types, etc.).

The synthesized method:

1. creates a `StringBuilder` instance,
2. appends the record name to the builder, followed by " { ",
3. invokes the record's `PrintMembers` method giving it the builder, followed by " " if it returned true,
4. appends "}",
5. returns the builder's contents with `builder.ToString()`.

For example, consider the following record types:

```
record R1(T1 P1);  
record R2(T1 P1, T2 P2, T3 P3) : R1(P1);
```

For those record types, the synthesized printing members would be something like:

```

class R1 : IEquatable<R1>
{
    public T1 P1 { get; init; }

    protected virtual bool PrintMembers(StringBuilder builder)
    {
        builder.Append(nameof(P1));
        builder.Append(" = ");
        builder.Append(this.P1); // or builder.Append(this.P1.ToString()); if P1 has a value type

        return true;
    }

    public override string ToString()
    {
        var builder = new StringBuilder();
        builder.Append(nameof(R1));
        builder.Append(" { ");

        if (PrintMembers(builder))
            builder.Append(" ");

        builder.Append("}");
        return builder.ToString();
    }
}

class R2 : R1, IEquatable<R2>
{
    public T2 P2 { get; init; }
    public T3 P3 { get; init; }

    protected override bool PrintMembers(StringBuilder builder)
    {
        if (base.PrintMembers(builder))
            builder.Append(", ");

        builder.Append(nameof(P2));
        builder.Append(" = ");
        builder.Append(this.P2); // or builder.Append(this.P2); if P2 has a value type

        builder.Append(", ");

        builder.Append(nameof(P3));
        builder.Append(" = ");
        builder.Append(this.P3); // or builder.Append(this.P3); if P3 has a value type

        return true;
    }

    public override string ToString()
    {
        var builder = new StringBuilder();
        builder.Append(nameof(R2));
        builder.Append(" { ");

        if (PrintMembers(builder))
            builder.Append(" ");

        builder.Append("}");
        return builder.ToString();
    }
}

```

Positional record members

In addition to the above members, records with a parameter list ("positional records") synthesize additional members with the same conditions as the members above.

Primary Constructor

A record type has a public constructor whose signature corresponds to the value parameters of the type declaration. This is called the primary constructor for the type, and causes the implicitly declared default class constructor, if present, to be suppressed. It is an error to have a primary constructor and a constructor with the same signature already present in the class.

At runtime the primary constructor

1. executes the instance initializers appearing in the class-body
2. invokes the base class constructor with the arguments provided in the `record_base` clause, if present

If a record has a primary constructor, any user-defined constructor, except "copy constructor" must have an explicit `this` constructor initializer.

Parameters of the primary constructor as well as members of the record are in scope within the `argument_list` of the `record_base` clause and within initializers of instance fields or properties. Instance members would be an error in these locations (similar to how instance members are in scope in regular constructor initializers today, but an error to use), but the parameters of the primary constructor would be in scope and useable and would shadow members. Static members would also be useable, similar to how base calls and initializers work in ordinary constructors today.

A warning is produced if a parameter of the primary constructor is not read.

Expression variables declared in the `argument_list` are in scope within the `argument_list`. The same shadowing rules as within an argument list of a regular constructor initializer apply.

Properties

For each record parameter of a record type declaration there is a corresponding public property member whose name and type are taken from the value parameter declaration.

For a record:

- A public `get` and `init` auto-property is created (see separate `init` accessor specification). An inherited `abstract` property with matching type is overridden. It is an error if the inherited property does not have `public` overridable `get` and `init` accessors. It is an error if the inherited property is hidden. The auto-property is initialized to the value of the corresponding primary constructor parameter. Attributes can be applied to the synthesized auto-property and its backing field by using `property:` or `field:` targets for attributes syntactically applied to the corresponding record parameter.

Deconstruct

A positional record with at least one parameter synthesizes a public void-returning instance method called `Deconstruct` with an out parameter declaration for each parameter of the primary constructor declaration. Each parameter of the `Deconstruct` method has the same type as the corresponding parameter of the primary constructor declaration. The body of the method assigns each parameter of the `Deconstruct` method to the value from an instance member access to a member of the same name. The method can be declared explicitly. It is an error if the explicit declaration does not match the expected signature or accessibility, or is static.

`with` expression

A `with` expression is a new expression using the following syntax.

```
with_expression
: switch_expression
| switch_expression 'with' '{' member_initializer_list? '}'
;

member_initializer_list
: member_initializer (',' member_initializer)*
;

member_initializer
: identifier '=' expression
;
```

A `with` expression is not permitted as a statement.

A `with` expression allows for "non-destructive mutation", designed to produce a copy of the receiver expression with modifications in assignments in the `member_initializer_list`.

A valid `with` expression has a receiver with a non-void type. The receiver type must be a record.

On the right hand side of the `with` expression is a `member_initializer_list` with a sequence of assignments to *identifier*, which must be an accessible instance field or property of the receiver's type.

First, receiver's "clone" method (specified above) is invoked and its result is converted to the receiver's type.

Then, each `member_initializer` is processed the same way as an assignment to a field or property access of the result of the conversion. Assignments are processed in lexical order.

Top-level statements

12/28/2021 • 4 minutes to read • [Edit Online](#)

Summary

Allow a sequence of *statements* to occur right before the *namespace_member_declarations* of a *compilation_unit* (i.e. source file).

The semantics are that if such a sequence of *statements* is present, the following type declaration, modulo the actual type name and the method name, would be emitted:

```
static class Program
{
    static async Task Main(string[] args)
    {
        // statements
    }
}
```

See also <https://github.com/dotnet/csharplang/issues/3117>.

Motivation

There's a certain amount of boilerplate surrounding even the simplest of programs, because of the need for an explicit `Main` method. This seems to get in the way of language learning and program clarity. The primary goal of the feature therefore is to allow C# programs without unnecessary boilerplate around them, for the sake of learners and the clarity of code.

Detailed design

Syntax

The only additional syntax is allowing a sequence of *statements* in a compilation unit, just before the *namespace_member_declarations*:

```
compilation_unit
: extern_alias_directive* using_directive* global_attributes? statement* namespace_member_declaration*
;
```

Only one *compilation_unit* is allowed to have *statements*.

Example:

```

if (args.Length == 0
    || !int.TryParse(args[0], out int n)
    || n < 0) return;
Console.WriteLine(Fib(n).curr);

(int curr, int prev) Fib(int i)
{
    if (i == 0) return (1, 0);
    var (curr, prev) = Fib(i - 1);
    return (curr + prev, curr);
}

```

Semantics

If any top-level statements are present in any compilation unit of the program, the meaning is as if they were combined in the block body of a `Main` method of a `Program` class in the global namespace, as follows:

```

static class Program
{
    static async Task Main(string[] args)
    {
        // statements
    }
}

```

Note that the names "Program" and "Main" are used only for illustrations purposes, actual names used by compiler are implementation dependent and neither the type, nor the method can be referenced by name from source code.

The method is designated as the entry point of the program. Explicitly declared methods that by convention could be considered as an entry point candidates are ignored. A warning is reported when that happens. It is an error to specify `-main:<type>` compiler switch when there are top-level statements.

The entry point method always has one formal parameter, `string[] args`. The execution environment creates and passes a `string[]` argument containing the command-line arguments that were specified when the application was started. The `string[]` argument is never null, but it may have a length of zero if no command-line arguments were specified. The 'args' parameter is in scope within top-level statements and is not in scope outside of them. Regular name conflict/shadowing rules apply.

Async operations are allowed in top-level statements to the degree they are allowed in statements within a regular async entry point method. However, they are not required, if `await` expressions and other async operations are omitted, no warning is produced.

The signature of the generated entry point method is determined based on operations used by the top level statements as follows:

ASYNC-OPERATIONS\RETURN-WITH-EXPRESSION	PRESENT	ABSENT
Present	<code>static Task<int> Main(string[] args)</code>	<code>static Task Main(string[] args)</code>
Absent	<code>static int Main(string[] args)</code>	<code>static void Main(string[] args)</code>

The example above would yield the following `$Main` method declaration:

```

static class $Program
{
    static void $Main(string[] args)
    {
        if (args.Length == 0
            || !int.TryParse(args[0], out int n)
            || n < 0) return;
        Console.WriteLine(Fib(n).curr);

        (int curr, int prev) Fib(int i)
        {
            if (i == 0) return (1, 0);
            var (curr, prev) = Fib(i - 1);
            return (curr + prev, curr);
        }
    }
}

```

At the same time an example like this:

```

await System.Threading.Tasks.Task.Delay(1000);
System.Console.WriteLine("Hi!");

```

would yield:

```

static class $Program
{
    static async Task $Main(string[] args)
    {
        await System.Threading.Tasks.Task.Delay(1000);
        System.Console.WriteLine("Hi!");
    }
}

```

An example like this:

```

await System.Threading.Tasks.Task.Delay(1000);
System.Console.WriteLine("Hi!");
return 0;

```

would yield:

```

static class $Program
{
    static async Task<int> $Main(string[] args)
    {
        await System.Threading.Tasks.Task.Delay(1000);
        System.Console.WriteLine("Hi!");
        return 0;
    }
}

```

And an example like this:

```

System.Console.WriteLine("Hi!");
return 2;

```


would yield:

```
static class $Program
{
    static int $Main(string[] args)
    {
        System.Console.WriteLine("Hi!");
        return 2;
    }
}
```

Scope of top-level local variables and local functions

Even though top-level local variables and functions are "wrapped" into the generated entry point method, they should still be in scope throughout the program in every compilation unit. For the purpose of simple-name evaluation, once the global namespace is reached:

- First, an attempt is made to evaluate the name within the generated entry point method and only if this attempt fails
- The "regular" evaluation within the global namespace declaration is performed.

This could lead to name shadowing of namespaces and types declared within the global namespace as well as to shadowing of imported names.

If the simple name evaluation occurs outside of the top-level statements and the evaluation yields a top-level local variable or function, that should lead to an error.

In this way we protect our future ability to better address "Top-level functions" (scenario 2 in <https://github.com/dotnet/csharplang/issues/3117>), and are able to give useful diagnostics to users who mistakenly believe them to be supported.

Nullable Reference Types Specification

12/28/2021 • 18 minutes to read • [Edit Online](#)

This is a work in progress - several parts are missing or incomplete.

This feature adds two new kinds of nullable types (nullable reference types and nullable generic types) to the existing nullable value types, and introduces a static flow analysis for purpose of null-safety.

Syntax

Nullable reference types and nullable type parameters

Nullable reference types and nullable type parameters have the same syntax `T?` as the short form of nullable value types, but do not have a corresponding long form.

For the purposes of the specification, the current `nullable_type` production is renamed to `nullable_value_type`, and `nullable_reference_type` and `nullable_type_parameter` productions are added:

```
type
    : value_type
    | reference_type
    | nullable_type_parameter
    | type_parameter
    | type_unsafe
    ;

reference_type
    : ...
    | nullable_reference_type
    ;

nullable_reference_type
    : non_nullable_reference_type '?'
    ;

non_nullable_reference_type
    : reference_type
    ;

nullable_type_parameter
    : non_nullable_non_value_type_parameter '?'
    ;

non_nullable_non_value_type_parameter
    : type_parameter
    ;
```

The `non_nullable_reference_type` in a `nullable_reference_type` must be a nonnullable reference type (class, interface, delegate or array).

The `non_nullable_non_value_type_parameter` in `nullable_type_parameter` must be a type parameter that isn't constrained to be a value type.

Nullable reference types and nullable type parameters cannot occur in the following positions:

- as a base class or interface
- as the receiver of a `member_access`

- as the `type` in an `object_creation_expression`
- as the `delegate_type` in a `delegate_creation_expression`
- as the `type` in an `is_expression`, a `catch_clause` or a `type_pattern`
- as the `interface` in a fully qualified interface member name

A warning is given on a `nullable_reference_type` and `nullable_type_parameter` in a *disabled* nullable annotation context.

`class` **and** `class?` **constraint**

The `class` constraint has a nullable counterpart `class?`:

```
primary_constraint
: ...
| 'class' '?'
;
```

A type parameter constrained with `class` (in an *enabled* annotation context) must be instantiated with a nonnullable reference type.

A type parameter constrained with `class?` (or `class` in a *disabled* annotation context) may either be instantiated with a nullable or nonnullable reference type.

A warning is given on a `class?` constraint in a *disabled* annotation context.

`notnull` **constraint**

A type parameter constrained with `notnull` may not be a nullable type (nullable value type, nullable reference type or nullable type parameter).

```
primary_constraint
: ...
| 'notnull'
;
```

`default` **constraint**

The `default` constraint can be used on a method override or explicit implementation to disambiguate `T?` meaning "nullable type parameter" from "nullable value type" (`Nullable<T>`). Lacking the `default` constraint a `T?` syntax in an override or explicit implementation will be interpreted as `Nullable<T>`.

See <https://github.com/dotnet/csharplang/blob/master/proposals/csharp-9.0/unconstrained-type-parameter-annotations.md#default-constraint>

The null-forgiving operator

The post-fix `!` operator is called the null-forgiving operator. It can be applied on a *primary_expression* or within a *null_conditional_expression*.

```

primary_expression
: ...
| null_forgiving_expression
;

null_forgiving_expression
: primary_expression '!'
;

null_conditional_expression
: primary_expression null_conditional_operations_no_suppression suppression?
;

null_conditional_operations_no_suppression
: null_conditional_operations? '?' '.' identifier type_argument_list?
| null_conditional_operations? '?' '[' argument_list ']'
| null_conditional_operations '.' identifier type_argument_list?
| null_conditional_operations '[' argument_list ']'
| null_conditional_operations '(' argument_list? ')'
;

null_conditional_operations
: null_conditional_operations_no_suppression suppression?
;

suppression
: '!'
;

```

For example:

```

var v = expr!;
expr!.M();
_ = a?.b!.c;

```

The postfix `!` operator has no runtime effect - it evaluates to the result of the underlying expression. Its only role is to change the null state of the expression to "not null", and to limit warnings given on its use.

Nullable compiler directives

`#nullable` directives control the nullable annotation and warning contexts.

```

pp_directive
: ...
| pp_nullable
;

pp_nullable
: whitespace? '#' whitespace? 'nullable' whitespace nullable_action (whitespace nullable_target)?
pp_new_line
;

nullable_action
: 'disable'
| 'enable'
| 'restore'
;

nullable_target
: 'warnings'
| 'annotations'
;

```

`#pragma warning` directives are expanded to allow changing the nullable warning context:

```
pragma_warning_body
: ...
| 'warning' whitespace warning_action whitespace 'nullable'
;
```

For example:

```
#pragma warning disable nullable
```

Nullable contexts

Every line of source code has a *nullable annotation context* and a *nullable warning context*. These control whether nullable annotations have effect, and whether nullability warnings are given. The annotation context of a given line is either *disabled* or *enabled*. The warning context of a given line is either *disabled* or *enabled*.

Both contexts can be specified at the project level (outside of C# source code), or anywhere within a source file via `#nullable` pre-processor directives. If no project level settings are provided the default is for both contexts to be *disabled*.

The `#nullable` directive controls the annotation and warning contexts within the source text, and take precedence over the project-level settings.

A directive sets the context(s) it controls for subsequent lines of code, until another directive overrides it, or until the end of the source file.

The effect of the directives is as follows:

- `#nullable disable` : Sets the nullable annotation and warning contexts to *disabled*
- `#nullable enable` : Sets the nullable annotation and warning contexts to *enabled*
- `#nullable restore` : Restores the nullable annotation and warning contexts to project settings
- `#nullable disable annotations` : Sets the nullable annotation context to *disabled*
- `#nullable enable annotations` : Sets the nullable annotation context to *enabled*
- `#nullable restore annotations` : Restores the nullable annotation context to project settings
- `#nullable disable warnings` : Sets the nullable warning context to *disabled*
- `#nullable enable warnings` : Sets the nullable warning context to *enabled*
- `#nullable restore warnings` : Restores the nullable warning context to project settings

Nullability of types

A given type can have one of three nullabilities: *oblivious*, *nonnullable*, and *nullable*.

Nonnullable types may cause warnings if a potential `null` value is assigned to them. *Oblivious* and *nullable* types, however, are "null-assignable" and can have `null` values assigned to them without warnings.

Values of *oblivious* and *nonnullable* types can be dereferenced or assigned without warnings. Values of *nullable* types, however, are "null-yielding" and may cause warnings when dereferenced or assigned without proper null checking.

The *default null state* of a null-yielding type is "maybe null" or "maybe default". The default null state of a non-null-yielding type is "not null".

The kind of type and the nullable annotation context it occurs in determine its nullability:

- A nonnullable value type `S` is always *nonnullable*
- A nullable value type `S?` is always *nullable*
- An unannotated reference type `C` in a *disabled* annotation context is *oblivious*
- An unannotated reference type `C` in an *enabled* annotation context is *nonnullable*
- A nullable reference type `C?` is *nullable* (but a warning may be yielded in a *disabled* annotation context)

Type parameters additionally take their constraints into account:

- A type parameter `T` where all constraints (if any) are either nullable types or the `class?` constraint is *nullable*
- A type parameter `T` where at least one constraint is either *oblivious* or *nonnullable* or one of the `struct` or `class` or `notnull` constraints is
 - *oblivious* in a *disabled* annotation context
 - *nonnullable* in an *enabled* annotation context
- A nullable type parameter `T?` is *nullable*, but a warning is yielded in a *disabled* annotation context if `T` isn't a value type

Oblivious vs nonnullable

A `type` is deemed to occur in a given annotation context when the last token of the type is within that context.

Whether a given reference type `C` in source code is interpreted as oblivious or nonnullable depends on the annotation context of that source code. But once established, it is considered part of that type, and "travels with it" e.g. during substitution of generic type arguments. It is as if there is an annotation like `?` on the type, but invisible.

Constraints

Nullable reference types can be used as generic constraints.

`class?` is a new constraint denoting "possibly nullable reference type", whereas `class` in an *enabled* annotation context denotes "nonnullable reference type".

`default` is a new constraint denoting a type parameter that isn't known to be a reference or value type. It can only be used on overridden and explicitly implemented methods. With this constraint, `T?` means a nullable type parameter, as opposed to being a shorthand for `Nullable<T>`.

`notnull` is a new constraint denoting a type parameter that is nonnullable.

The nullability of a type argument or of a constraint does not impact whether the type satisfies the constraint, except where that is already the case today (nullable value types do not satisfy the `struct` constraint). However, if the type argument does not satisfy the nullability requirements of the constraint, a warning may be given.

Null state and null tracking

Every expression in a given source location has a *null state*, which indicated whether it is believed to potentially evaluate to null. The null state is either "not null", "maybe null", or "maybe default". The null state is used to determine whether a warning should be given about null-unsafe conversions and dereferences.

The distinction between "maybe null" and "maybe default" is subtle and applies to type parameters. The distinction is that a type parameter `T` which has the state "maybe null" means the value is in the domain of legal values for `T` however that legal value may include `null`. Whereas a "maybe default" means that the value may be outside the legal domain of values for `T`.

Example:

```
// The value `t` here has the state "maybe null". It's possible for `T` to be instantiated
// with `string?` in which case `null` would be within the domain of legal values here. The
// assumption though is the value provided here is within the legal values of `T`. Hence
// if `T` is `string` then `null` will not be a value, just as we assume that `null` is not
// provided for a normal `string` parameter
void M<T>(T t)
{
    // There is no guarantee that default(T) is within the legal values for T hence the
    // state *must* be "maybe-default" and hence `local` must be `T?`
    T? local = default(T);
}
```

Null tracking for variables

For certain expressions denoting variables, fields or properties, the null state is tracked between occurrences, based on assignments to them, tests performed on them and the control flow between them. This is similar to how definite assignment is tracked for variables. The tracked expressions are the ones of the following form:

```
tracked_expression
: simple_name
| this
| base
| tracked_expression '.' identifier
;
```

Where the identifiers denote fields or properties.

The null state for tracked variables is "not null" in unreachable code. This follows other decisions around unreachable code like considering all locals to be definitely assigned.

Describe null state transitions similar to definite assignment

Null state for expressions

The null state of an expression is derived from its form and type, and from the null state of variables involved in it.

Literals

The null state of a `null` literal depends on the target type of the expression. If the target type is a type parameter constrained to a reference type then it's "maybe default". Otherwise it is "maybe null".

The null state of a `default` literal depends on the target type of the `default` literal. A `default` literal with target type `T` has the same null state as the `default(T)` expression.

The null state of any other literal is "not null".

Simple names

If a `simple_name` is not classified as a value, its null state is "not null". Otherwise it is a tracked expression, and its null state is its tracked null state at this source location.

Member access

If a `member_access` is not classified as a value, its null state is "not null". Otherwise, if it is a tracked expression, its null state is its tracked null state at this source location. Otherwise its null state is the default null state of its type.

```

var person = new Person();

// The receiver is a tracked expression hence the member_access of the property
// is tracked as well
if (person.FirstName is not null)
{
    Use(person.FirstName);
}

// The return of an invocation is not a tracked expression hence the member_access
// of the return is also not tracked
if (GetAnonymous().FirstName is not null)
{
    // Warning: Cannot convert null literal to non-nullable reference type.
    Use(GetAnonymous().FirstName);
}

void Use(string s)
{
    // ...
}

public class Person
{
    public string? FirstName { get; set; }
    public string? LastName { get; set; }

    private static Person s_anonymous = new Person();
    public static Person GetAnonymous() => s_anonymous;
}

```

Invocation expressions

If an `invocation_expression` invokes a member that is declared with one or more attributes for special null behavior, the null state is determined by those attributes. Otherwise the null state of the expression is the default null state of its type.

The null state of an `invocation_expression` is not tracked by the compiler.

```

// The result of an invocation_expression is not tracked
if (GetText() is not null)
{
    // Warning: Converting null literal or possible null value to non-nullable type.
    string s = GetText();
    // Warning: Dereference of a possibly null reference.
    Use(s);
}

// Nullable friendly pattern
if (GetText() is string s)
{
    Use(s);
}

string? GetText() => ...
Use(string s) { }

```

Element access

If an `element_access` invokes an indexer that is declared with one or more attributes for special null behavior, the null state is determined by those attributes. Otherwise the null state of the expression is the default null state of its type.


```

object?[] array = ...;
if (array[0] != null)
{
    // Warning: Converting null literal or possible null value to non-nullable type.
    object o = array[0];
    // Warning: Dereference of a possibly null reference.
    Console.WriteLine(o.ToString());
}

// Nullable friendly pattern
if (array[0] is {} o)
{
    Console.WriteLine(o.ToString());
}

```

Base access

If `B` denotes the base type of the enclosing type, `base.I` has the same null state as `((B)this).I` and `base[E]` has the same null state as `((B)this)[E]`.

Default expressions

`default(T)` has the null state based on the properties of the type `T`:

- If the type is a *nonnullable* type then it has the null state "not null"
- Else if the type is a type parameter then it has the null state "maybe default"
- Else it has the null state "maybe null"

Null-conditional expressions ?.

A `null_conditional_expression` has the null state based on the expression type. Note that this refers to the type of the `null_conditional_expression`, not the original type of the member being invoked:

- If the type is a *nullable* value type then it has the null state "maybe null"
- Else if the type is a *nullable* type parameter then it has the null state "maybe default"
- Else it has the null state "maybe null"

Cast expressions

If a cast expression `(T)E` invokes a user-defined conversion, then the null state of the expression is the default null state for the type of the user-defined conversion. Otherwise:

- If `T` is a *nonnullable* value type then `T` has the null state "not null"
- Else if `T` is a *nullable* value type then `T` has the null state "maybe null"
- Else if `T` is a *nullable* type in the form `U?` where `U` is a type parameter then `T` has the null state "maybe default"
- Else if `T` is a *nullable* type, and `E` has null state "maybe null" or "maybe default", then `T` has the null state "maybe null"
- Else if `T` is a type parameter, and `E` has null state "maybe null" or "maybe default", then `T` has the null state "maybe default"
- Else `T` has the same null state as `E`

Unary and binary operators

If a unary or binary operator invokes an user-defined operator then the null state of the expression is the default null state for the type of the user-defined operator. Otherwise it is the null state of the expression.

Something special to do for binary `+` over strings and delegates?

Await expressions

The null state of `await E` is the default null state of its type.

The `as` operator

The null state of an `E as T` expression depends first on properties of the type `T`. If the type of `T` is *nonnullable* then the null state is "not null". Otherwise the null state depends on the conversion from the type of `E` to type `T`:

- If the conversion is an identity, boxing, implicit reference, or implicit nullable conversion, then the null state is the null state of `E`
- Else if `T` is a type parameter then it has the null state "maybe default"
- Else it has the null state "maybe null"

The null-coalescing operator

The null state of `E1 ?? E2` is the null state of `E2`

The conditional operator

The null state of `E1 ? E2 : E3` is based on the null state of `E2` and `E3`:

- If both are "not null" then the null state is "not null"
- Else if either is "maybe default" then the null state is "maybe default"
- Else the null state is "not null"

Query expressions

The null state of a query expression is the default null state of its type.

Additional work needed here

Assignment operators

`E1 = E2` and `E1 op= E2` have the same null state as `E2` after any implicit conversions have been applied.

Expressions that propagate null state

`(E)`, `checked(E)` and `unchecked(E)` all have the same null state as `E`.

Expressions that are never null

The null state of the following expression forms is always "not null":

- `this` access
- interpolated strings
- `new` expressions (object, delegate, anonymous object and array creation expressions)
- `typeof` expressions
- `nameof` expressions
- anonymous functions (anonymous methods and lambda expressions)
- null-forgiving expressions
- `is` expressions

Nested functions

Nested functions (lambdas and local functions) are treated like methods, except in regards to their captured variables. The initial state of a captured variable inside a lambda or local function is the intersection of the nullable state of the variable at all the "uses" of that nested function or lambda. A use of a local function is either a call to that function, or where it is converted to a delegate. A use of a lambda is the point at which it is defined in source.

Type inference

nullable implicitly typed local variables

`var` infers an annotated type for reference types, and type parameters that aren't constrained to be a value type. For instance:

- in `var s = ""`; the `var` is inferred as `string?`.
- in `var t = new T()`; with an unconstrained `T` the `var` is inferred as `T?`.

Generic type inference

Generic type inference is enhanced to help decide whether inferred reference types should be nullable or not. This is a best effort. It may yield warnings regarding nullability constraints, and may lead to nullable warnings when the inferred types of the selected overload are applied to the arguments.

The first phase

Nullable reference types flow into the bounds from the initial expressions, as described below. In addition, two new kinds of bounds, namely `null` and `default` are introduced. Their purpose is to carry through occurrences of `null` or `default` in the input expressions, which may cause an inferred type to be nullable, even when it otherwise wouldn't.

The determination of what bounds to add in the first phase are enhanced as follows:

If an argument `Ei` has a reference type, the type `U` used for inference depends on the null state of `Ei` as well as its declared type:

- If the declared type is a nonnullable reference type `U0` or a nullable reference type `U0?` then
 - if the null state of `Ei` is "not null" then `U` is `U0`
 - if the null state of `Ei` is "maybe null" then `U` is `U0?`
- Otherwise if `Ei` has a declared type, `U` is that type
- Otherwise if `Ei` is `null` then `U` is the special bound `null`
- Otherwise if `Ei` is `default` then `U` is the special bound `default`
- Otherwise no inference is made.

Exact, upper-bound and lower-bound inferences

In inferences *from* the type `U` *to* the type `V`, if `V` is a nullable reference type `V0?`, then `V0` is used instead of `V` in the following clauses.

- If `V` is one of the unfixed type variables, `U` is added as an exact, upper or lower bound as before
- Otherwise, if `U` is `null` or `default`, no inference is made
- Otherwise, if `U` is a nullable reference type `U0?`, then `U0` is used instead of `U` in the subsequent clauses.

The essence is that nullability that pertains directly to one of the unfixed type variables is preserved into its bounds. For the inferences that recurse further into the source and target types, on the other hand, nullability is ignored. It may or may not match, but if it doesn't, a warning will be issued later if the overload is chosen and applied.

Fixing

The spec currently does not do a good job of describing what happens when multiple bounds are identity convertible to each other, but are different. This may happen between `object` and `dynamic`, between tuple types that differ only in element names, between types constructed thereof and now also between `C` and `C?` for reference types.

In addition we need to propagate "nullness" from the input expressions to the result type.

To handle these we add more phases to fixing, which is now:

1. Gather all the types in all the bounds as candidates, removing `?` from all that are nullable reference types

2. Eliminate candidates based on requirements of exact, lower and upper bounds (keeping `null` and `default` bounds)
3. Eliminate candidates that do not have an implicit conversion to all the other candidates
4. If the remaining candidates do not all have identity conversions to one another, then type inference fails
5. *Merge* the remaining candidates as described below
6. If the resulting candidate is a reference type and *all* of the exact bounds or *any* of the lower bounds are nullable reference types, `null` or `default`, then `?` is added to the resulting candidate, making it a nullable reference type.

Merging is described between two candidate types. It is transitive and commutative, so the candidates can be merged in any order with the same ultimate result. It is undefined if the two candidate types are not identity convertible to each other.

The *Merge* function takes two candidate types and a direction (+ or -):

- $\text{Merge}(T, T, d) = T$
- $\text{Merge}(S, T?, +) = \text{Merge}(S?, T, +) = \text{Merge}(S, T, +)?$
- $\text{Merge}(S, T?, -) = \text{Merge}(S?, T, -) = \text{Merge}(S, T, -)$
- $\text{Merge}(C\langle S_1, \dots, S_n \rangle, C\langle T_1, \dots, T_n \rangle, +) = C\langle \text{Merge}(S_1, T_1, d_1), \dots, \text{Merge}(S_n, T_n, d_n) \rangle$, where
 - $d_i = +$ if the i 'th type parameter of $C\langle \dots \rangle$ is covariant
 - $d_i = -$ if the i 'th type parameter of $C\langle \dots \rangle$ is contra- or invariant
- $\text{Merge}(C\langle S_1, \dots, S_n \rangle, C\langle T_1, \dots, T_n \rangle, -) = C\langle \text{Merge}(S_1, T_1, d_1), \dots, \text{Merge}(S_n, T_n, d_n) \rangle$, where
 - $d_i = -$ if the i 'th type parameter of $C\langle \dots \rangle$ is covariant
 - $d_i = +$ if the i 'th type parameter of $C\langle \dots \rangle$ is contra- or invariant
- $\text{Merge}(S_1\ s_1, \dots, S_n\ s_n, (T_1\ t_1, \dots, T_n\ t_n), d) = (\text{Merge}(S_1, T_1, d)_{n_1}, \dots, \text{Merge}(S_n, T_n, d)_{n_n})$, where
 - n_i is absent if s_i and t_i differ, or if both are absent
 - n_i is s_i if s_i and t_i are the same
- $\text{Merge}(\text{object}, \text{dynamic}) = \text{Merge}(\text{dynamic}, \text{object}) = \text{dynamic}$

Warnings

Potential null assignment

Potential null dereference

Constraint nullability mismatch

Nullable types in disabled annotation context

Override and implementation nullability mismatch

Attributes for special null behavior

Pattern-matching changes for C# 9.0

12/28/2021 • 11 minutes to read • [Edit Online](#)

We are considering a small handful of enhancements to pattern-matching for C# 9.0 that have natural synergy and work well to address a number of common programming problems:

- <https://github.com/dotnet/csharpplang/issues/2925> Type patterns
- <https://github.com/dotnet/csharpplang/issues/1350> Parenthesized patterns to enforce or emphasize precedence of the new combinators
- <https://github.com/dotnet/csharpplang/issues/1350> Conjunctive `and` patterns that require both of two different patterns to match;
- <https://github.com/dotnet/csharpplang/issues/1350> Disjunctive `or` patterns that require either of two different patterns to match;
- <https://github.com/dotnet/csharpplang/issues/1350> Negated `not` patterns that require a given pattern *not* to match; and
- <https://github.com/dotnet/csharpplang/issues/812> Relational patterns that require the input value to be less than, less than or equal to, etc a given constant.

Parenthesized Patterns

Parenthesized patterns permit the programmer to put parentheses around any pattern. This is not so useful with the existing patterns in C# 8.0, however the new pattern combinators introduce a precedence that the programmer may want to override.

```
primary_pattern
    : parenthesized_pattern
    | // all of the existing forms
    ;
parenthesized_pattern
    : '(' pattern ')'
```

Type Patterns

We permit a type as a pattern:

```
primary_pattern
    : type-pattern
    | // all of the existing forms
    ;
type_pattern
    : type
```

This retcons the existing *is-type-expression* to be an *is-pattern-expression* in which the pattern is a *type-pattern*, though we would not change the syntax tree produced by the compiler.

One subtle implementation issue is that this grammar is ambiguous. A string such as `a.b` can be parsed either as a qualified name (in a type context) or a dotted expression (in an expression context). The compiler is already capable of treating a qualified name the same as a dotted expression in order to handle something like

`e is Color.Red`. The compiler's semantic analysis would be further extended to be capable of binding a (syntactic) constant pattern (e.g. a dotted expression) as a type in order to treat it as a bound type pattern in order to support this construct.

After this change, you would be able to write

```
void M(object o1, object o2)
{
    var t = (o1, o2);
    if (t is (int, string)) {} // test if o1 is an int and o2 is a string
    switch (o1) {
        case int: break; // test if o1 is an int
        case System.String: break; // test if o1 is a string
    }
}
```

Relational Patterns

Relational patterns permit the programmer to express that an input value must satisfy a relational constraint when compared to a constant value:

```
public static LifeStage LifeStageAtAge(int age) => age switch
{
    < 0 => LifeStage.Prenatal,
    < 2 => LifeStage.Infant,
    < 4 => LifeStage.Toddler,
    < 6 => LifeStage.EarlyChild,
    < 12 => LifeStage.MiddleChild,
    < 20 => LifeStage.Adolescent,
    < 40 => LifeStage.EarlyAdult,
    < 65 => LifeStage.MiddleAdult,
    _ => LifeStage.LateAdult,
};
```

Relational patterns support the relational operators `<`, `<=`, `>`, and `>=` on all of the built-in types that support such binary relational operators with two operands of the same type in an expression. Specifically, we support all of these relational patterns for `sbyte`, `byte`, `short`, `ushort`, `int`, `uint`, `long`, `ulong`, `char`, `float`, `double`, `decimal`, `nint`, and `nuint`.

```
primary_pattern
    : relational_pattern
    ;
relational_pattern
    : '<' relational_expression
    | '<=' relational_expression
    | '>' relational_expression
    | '>=' relational_expression
    ;
```

The expression is required to evaluate to a constant value. It is an error if that constant value is `double.NaN` or `float.NaN`. It is an error if the expression is a null constant.

When the input is a type for which a suitable built-in binary relational operator is defined that is applicable with the input as its left operand and the given constant as its right operand, the evaluation of that operator is taken as the meaning of the relational pattern. Otherwise we convert the input to the type of the expression using an explicit nullable or unboxing conversion. It is a compile-time error if no such conversion exists. The pattern is considered not to match if the conversion fails. If the conversion succeeds then the result of the pattern-

matching operation is the result of evaluating the expression `e OP v` where `e` is the converted input, `OP` is the relational operator, and `v` is the constant expression.

Pattern Combinators

Pattern *combinators* permit matching both of two different patterns using `and` (this can be extended to any number of patterns by the repeated use of `and`), either of two different patterns using `or` (ditto), or the *negation* of a pattern using `not`.

A common use of a combinator will be the idiom

```
if (e is not null) ...
```

More readable than the current idiom `e is object`, this pattern clearly expresses that one is checking for a non-null value.

The `and` and `or` combinators will be useful for testing ranges of values

```
bool IsLetter(char c) => c is >= 'a' and <= 'z' or >= 'A' and <= 'Z';
```

This example illustrates that `and` will have a higher parsing priority (i.e. will bind more closely) than `or`. The programmer can use the *parenthesized pattern* to make the precedence explicit:

```
bool IsLetter(char c) => c is (>= 'a' and <= 'z') or (>= 'A' and <= 'Z');
```

Like all patterns, these combinators can be used in any context in which a pattern is expected, including nested patterns, the *is-pattern-expression*, the *switch-expression*, and the pattern of a switch statement's case label.

```
pattern
: disjunctive_pattern
;
disjunctive_pattern
: disjunctive_pattern 'or' conjunctive_pattern
| conjunctive_pattern
;
conjunctive_pattern
: conjunctive_pattern 'and' negated_pattern
| negated_pattern
;
negated_pattern
: 'not' negated_pattern
| primary_pattern
;
primary_pattern
: // all of the patterns forms previously defined
;
```

Change to 7.5.4.2 Grammar Ambiguities

Due to the introduction of the *type pattern*, it is possible for a generic type to appear before the token `=>`. We therefore add `=>` to the set of tokens listed in *7.5.4.2 Grammar Ambiguities* to permit disambiguation of the `<` that begins the type argument list. See also <https://github.com/dotnet/roslyn/issues/47614>.

Open Issues with Proposed Changes

Syntax for relational operators

Are `and`, `or`, and `not` some kind of contextual keyword? If so, is there a breaking change (e.g. compared to their use as a designator in a *declaration-pattern*).

Semantics (e.g. type) for relational operators

We expect to support all of the primitive types that can be compared in an expression using a relational operator. The meaning in simple cases is clear

```
bool IsValidPercentage(int x) => x is >= 0 and <= 100;
```

But when the input is not such a primitive type, what type do we attempt to convert it to?

```
bool IsValidPercentage(object x) => x is >= 0 and <= 100;
```

We have proposed that when the input type is already a comparable primitive, that is the type of the comparison. However, when the input is not a comparable primitive, we treat the relational as including an implicit type test to the type of the constant on the right-hand-side of the relational. If the programmer intends to support more than one input type, that must be done explicitly:

```
bool IsValidPercentage(object x) => x is
    >= 0 and <= 100 or    // integer tests
    >= 0F and <= 100F or // float tests
    >= 0D and <= 100D;    // double tests
```

Flowing type information from the left to the right of `and`

It has been suggested that when you write an `and` combinator, type information learned on the left about the top-level type could flow to the right. For example

```
bool isSmallByte(object o) => o is byte and < 100;
```

Here, the *input type* to the second pattern is narrowed by the *type narrowing* requirements of left of the `and`. We would define type narrowing semantics for all patterns as follows. The *narrowed type* of a pattern `P` is defined as follows:

1. If `P` is a type pattern, the *narrowed type* is the type of the type pattern's type.
2. If `P` is a declaration pattern, the *narrowed type* is the type of the declaration pattern's type.
3. If `P` is a recursive pattern that gives an explicit type, the *narrowed type* is that type.
4. If `P` is [matched via the rules for](#) `ITuple`, the *narrowed type* is the type `System.Runtime.CompilerServices.ITuple`.
5. If `P` is a constant pattern where the constant is not the null constant and where the expression has no *constant expression conversion* to the *input type*, the *narrowed type* is the type of the constant.
6. If `P` is a relational pattern where the constant expression has no *constant expression conversion* to the *input type*, the *narrowed type* is the type of the constant.
7. If `P` is an `or` pattern, the *narrowed type* is the common type of the *narrowed type* of the subpatterns if such a common type exists. For this purpose, the common type algorithm considers only identity, boxing, and implicit reference conversions, and it considers all subpatterns of a sequence of `or` patterns (ignoring parenthesized patterns).
8. If `P` is an `and` pattern, the *narrowed type* is the *narrowed type* of the right pattern. Moreover, the *narrowed type* of the left pattern is the *input type* of the right pattern.
9. Otherwise the *narrowed type* of `P` is `P`'s input type.

Variable definitions and definite assignment

The addition of `or` and `not` patterns creates some interesting new problems around pattern variables and definite assignment. Since variables can normally be declared at most once, it would seem any pattern variable declared on one side of an `or` pattern would not be definitely assigned when the pattern matches. Similarly, a variable declared inside a `not` pattern would not be expected to be definitely assigned when the pattern matches. The simplest way to address this is to forbid declaring pattern variables in these contexts. However, this may be too restrictive. There are other approaches to consider.

One scenario that is worth considering is this

```
if (e is not int i) return;
M(i); // is i definitely assigned here?
```

This does not work today because, for an *is-pattern-expression*, the pattern variables are considered *definitely assigned* only where the *is-pattern-expression* is true ("definitely assigned when true").

Supporting this would be simpler (from the programmer's perspective) than also adding support for a negated-condition `if` statement. Even if we add such support, programmers would wonder why the above snippet does not work. On the other hand, the same scenario in a `switch` makes less sense, as there is no corresponding point in the program where *definitely assigned when false* would be meaningful. Would we permit this in an *is-pattern-expression* but not in other contexts where patterns are permitted? That seems irregular.

Related to this is the problem of definite assignment in a *disjunctive-pattern*.

```
if (e is 0 or int i)
{
    M(i); // is i definitely assigned here?
}
```

We would only expect `i` to be definitely assigned when the input is not zero. But since we don't know whether the input is zero or not inside the block, `i` is not definitely assigned. However, what if we permit `i` to be declared in different mutually exclusive patterns?

```
if ((e1, e2) is (0, int i) or (int i, 0))
{
    M(i);
}
```

Here, the variable `i` is definitely assigned inside the block, and takes its value from the other element of the tuple when a zero element is found.

It has also been suggested to permit variables to be (multiply) defined in every case of a case block:

```
case (0, int x):
case (int x, 0):
    Console.WriteLine(x);
```

To make any of this work, we would have to carefully define where such multiple definitions are permitted and under what conditions such a variable is considered definitely assigned.

Should we elect to defer such work until later (which I advise), we could say in C# 9

- beneath a `not` or `or`, pattern variables may not be declared.

Then, we would have time to develop some experience that would provide insight into the possible value of

relaxing that later.

Diagnostics, subsumption, and exhaustiveness

These new pattern forms introduce many new opportunities for diagnosable programmer error. We will need to decide what kinds of errors we will diagnose, and how to do so. Here are some examples:

```
case >= 0 and <= 100D:
```

This case can never match (because the input cannot be both an `int` and a `double`). We already have an error when we detect a case that can never match, but its wording ("The switch case has already been handled by a previous case" and "The pattern has already been handled by a previous arm of the switch expression") may be misleading in new scenarios. We may have to modify the wording to just say that the pattern will never match the input.

```
case 1 and 2:
```

Similarly, this would be an error because a value cannot be both `1` and `2`.

```
case 1 or 2 or 3 or 1:
```

This case is possible to match, but the `or 1` at the end adds no meaning to the pattern. I suggest we should aim to produce an error whenever some conjunct or disjunct of a compound pattern does not either define a pattern variable or affect the set of matched values.

```
case < 2: break;  
case 0 or 1 or 2 or 3 or 4 or 5: break;
```

Here, `0 or 1 or` adds nothing to the second case, as those values would have been handled by the first case. This too deserves an error.

```
byte b = ...;  
int x = b switch { <100 => 0, 100 => 1, 101 => 2, >101 => 3 };
```

A switch expression such as this should be considered *exhaustive* (it handles all possible input values).

In C# 8.0, a switch expression with an input of type `byte` is only considered exhaustive if it contains a final arm whose pattern matches everything (a *discard-pattern* or *var-pattern*). Even a switch expression that has an arm for every distinct `byte` value is not considered exhaustive in C# 8. In order to properly handle exhaustiveness of relational patterns, we will have to handle this case too. This will technically be a breaking change, but no user is likely to notice.

Init Only Setters

12/28/2021 • 15 minutes to read • [Edit Online](#)

Summary

This proposal adds the concept of init only properties and indexers to C#. These properties and indexers can be set at the point of object creation but become effectively `get` only once object creation has completed. This allows for a much more flexible immutable model in C#.

Motivation

The underlying mechanisms for building immutable data in C# haven't changed since 1.0. They remain:

1. Declaring fields as `readonly`.
2. Declaring properties that contain only a `get` accessor.

These mechanisms are effective at allowing the construction of immutable data but they do so by adding cost to the boilerplate code of types and opting such types out of features like object and collection initializers. This means developers must choose between ease of use and immutability.

A simple immutable object like `Point` requires twice as much boiler plate code to support construction as it does to declare the type. The bigger the type the bigger the cost of this boiler plate:

```
struct Point
{
    public int X { get; }
    public int Y { get; }

    public Point(int x, int y)
    {
        this.X = x;
        this.Y = y;
    }
}
```

The `init` accessor makes immutable objects more flexible by allowing the caller to mutate the members during the act of construction. That means the object's immutable properties can participate in object initializers and thus removes the need for all constructor boilerplate in the type. The `Point` type is now simply:

```
struct Point
{
    public int X { get; init; }
    public int Y { get; init; }
}
```

The consumer can then use object initializers to create the object

```
var p = new Point() { X = 42, Y = 13 };
```

Detailed Design

init accessors

An init only property (or indexer) is declared by using the `init` accessor in place of the `set` accessor:

```
class Student
{
    public string FirstName { get; init; }
    public string LastName { get; init; }
}
```

An instance property containing an `init` accessor is considered settable in the following circumstances, except when in a local function or lambda:

- During an object initializer
- During a `with` expression initializer
- Inside an instance constructor of the containing or derived type, on `this` or `base`
- Inside the `init` accessor of any property, on `this` or `base`
- Inside attribute usages with named parameters

The times above in which the `init` accessors are settable are collectively referred to in this document as the construction phase of the object.

This means the `Student` class can be used in the following ways:

```
var s = new Student()
{
    FirstName = "Jared",
    LastName = "Parosns",
};
s.LastName = "Parsons"; // Error: LastName is not settable
```

The rules around when `init` accessors are settable extend across type hierarchies. If the member is accessible and the object is known to be in the construction phase then the member is settable. That specifically allows for the following:

```
class Base
{
    public bool Value { get; init; }
}

class Derived : Base
{
    Derived()
    {
        // Not allowed with get only properties but allowed with init
        Value = true;
    }
}

class Consumption
{
    void Example()
    {
        var d = new Derived() { Value = true; };
    }
}
```

At the point an `init` accessor is invoked, the instance is known to be in the open construction phase. Hence an

`init` accessor is allowed to take the following actions in addition to what a normal `set` accessor can do:

1. Call other `init` accessors available through `this` or `base`
2. Assign `readonly` fields declared on the same type through `this`

```
class Complex
{
    readonly int Field1;
    int Field2;
    int Prop1 { get; init ; }
    int Prop2
    {
        get => 42;
        init
        {
            Field1 = 13; // okay
            Field2 = 13; // okay
            Prop1 = 13; // okay
        }
    }
}
```

The ability to assign `readonly` fields from an `init` accessor is limited to those fields declared on the same type as the accessor. It cannot be used to assign `readonly` fields in a base type. This rule ensures that type authors remain in control over the mutability behavior of their type. Developers who do not wish to utilize `init` cannot be impacted from other types choosing to do so:

```
class Base
{
    internal readonly int Field;
    internal int Property
    {
        get => Field;
        init => Field = value; // Okay
    }

    internal int OtherProperty { get; init; }
}

class Derived : Base
{
    internal readonly int DerivedField;
    internal int DerivedProperty
    {
        get => DerivedField;
        init
        {
            DerivedField = 42; // Okay
            Property = 0;      // Okay
            Field = 13;        // Error Field is readonly
        }
    }

    public Derived()
    {
        Property = 42; // Okay
        Field = 13;    // Error Field is readonly
    }
}
```

When `init` is used in a virtual property then all the overrides must also be marked as `init`. Likewise it is not possible to override a simple `set` with `init`.

```

class Base
{
    public virtual int Property { get; init; }
}

class C1 : Base
{
    public override int Property { get; init; }
}

class C2 : Base
{
    // Error: Property must have init to override Base.Property
    public override int Property { get; set; }
}

```

An `interface` declaration can also participate in `init` style initialization via the following pattern:

```

interface IPerson
{
    string Name { get; init; }
}

class Init
{
    void M<T>() where T : IPerson, new()
    {
        var local = new T()
        {
            Name = "Jared"
        };
        local.Name = "Jraed"; // Error
    }
}

```

Restrictions of this feature:

- The `init` accessor can only be used on instance properties
- A property cannot contain both an `init` and `set` accessor
- All overrides of a property must have `init` if the base had `init`. This rule also applies to interface implementation.

Readonly structs

`init` accessors (both auto-implemented accessors and manually-implemented accessors) are permitted on properties of `readonly struct`s, as well as `readonly` properties. `init` accessors are not permitted to be marked `readonly` themselves, in both `readonly` and non-`readonly` `struct` types.

```

readonly struct ReadonlyStruct1
{
    public int Prop1 { get; init; } // Allowed
}

struct ReadonlyStruct2
{
    public readonly int Prop2 { get; init; } // Allowed

    public int Prop3 { get; readonly init; } // Error
}

```

Metadata encoding

Property `init` accessors will be emitted as a standard `set` accessor with the return type marked with a modreq of `IsExternalInit`. This is a new type which will have the following definition:

```
namespace System.Runtime.CompilerServices
{
    public sealed class IsExternalInit
    {
    }
}
```

The compiler will match the type by full name. There is no requirement that it appear in the core library. If there are multiple types by this name then the compiler will tie break in the following order:

1. The one defined in the project being compiled
2. The one defined in corelib

If neither of these exist then a type ambiguity error will be issued.

The design for `IsExternalInit` is further covered in [this issue](#)

Questions

Breaking changes

One of the main pivot points in how this feature is encoded will come down to the following question:

Is it a binary breaking change to replace `init` with `set`?

Replacing `init` with `set` and thus making a property fully writable is never a source breaking change on a non-virtual property. It simply expands the set of scenarios where the property can be written. The only behavior in question is whether or not this remains a binary breaking change.

If we want to make the change of `init` to `set` a source and binary compatible change then it will force our hand on the modreq vs. attributes decision below because it will rule out modreqs as a solution. If on the other hand this is seen as a non-interesting then this will make the modreq vs. attribute decision less impactful.

Resolution This scenario is not seen as compelling by LDM.

Modreqs vs. attributes

The emit strategy for `init` property accessors must choose between using attributes or modreqs when emitting during metadata. These have different trade offs that need to be considered.

Annotating a property set accessor with a modreq declaration means CLI compliant compilers will ignore the accessor unless it understands the modreq. That means only compilers aware of `init` will read the member. Compilers unaware of `init` will ignore the `set` accessor and hence will not accidentally treat the property as read / write.

The downside of modreq is `init` becomes a part of the binary signature of the `set` accessor. Adding or removing `init` will break binary compatibility of the application.

Using attributes to annotate the `set` accessor means that only compilers which understand the attribute will know to limit access to it. A compiler unaware of `init` will see it as a simple read / write property and allow access.

This would seemingly mean this decision is a choice between extra safety at the expense of binary compatibility. Digging in a bit the extra safety is not exactly what it seems. It will not protect against the following circumstances:

1. Reflection over `public` members
2. The use of `dynamic`
3. Compilers that don't recognize modreqs

It should also be considered that, when we complete the IL verification rules for .NET 5, `init` will be one of those rules. That means extra enforcement will be gained from simply verifying compilers emitting verifiable IL.

The primary languages for .NET (C#, F# and VB) will all be updated to recognize these `init` accessors. Hence the only realistic scenario here is when a C# 9 compiler emits `init` properties and they are seen by an older toolset such as C# 8, VB 15, etc ... C# 8. That is the trade off to consider and weigh against binary compatibility.

Note This discussion primarily applies to members only, not to fields. While `init` fields were rejected by LDM they are still interesting to consider for the modreq vs. attribute discussion. The `init` feature for fields is a relaxation of the existing restriction of `readonly`. That means if we emit the fields as `readonly` + an attribute there is no risk of older compilers mis-using the field because they would already recognize `readonly`. Hence using a modreq here doesn't add any extra protection.

Resolution The feature will use a modreq to encode the property `init` setter. The compelling factors were (in no particular order):

- Desire to discourage older compilers from violating `init` semantics
- Desire to make adding or removing `init` in a `virtual` declaration or `interface` both a source and binary breaking change.

Given there was also no significant support for removing `init` to be a binary compatible change it made the choice of using modreq straight forward.

init vs. initonly

There were three syntax forms which got significant consideration during our LDM meeting:

```
// 1. Use init
int Option1 { get; init; }
// 2. Use init set
int Option2 { get; init set; }
// 3. Use initonly
int Option3 { get; initonly; }
```

Resolution There was no syntax which was overwhelmingly favored in LDM.

One point which got significant attention was how the choice of syntax would impact our ability to do `init` members as a general feature in the future. Choosing option 1 would mean that it would be difficult to define a property which had an `init` style `get` method in the future. Eventually it was decided that if we decided to go forward with general `init` members in future, we could allow `init` to be a modifier in the property accessor list as well as a short hand for `init set`. Essentially the following two declarations would be identical.

```
int Property1 { get; init; }
int Property1 { get; init set; }
```

The decision was made to move forward with `init` as a standalone accessor in the property accessor list.

Warn on failed init

Consider the following scenario. A type declares an `init` only member which is not set in the constructor. Should the code which constructs the object get a warning if they failed to initialize the value?

At that point it is clear the field will never be set and hence has a lot of similarities with the warning around

failing to initialize `private` data. Hence a warning would seemingly have some value here?

There are significant downsides to this warning though:

1. It complicates the compatibility story of changing `readonly` to `init`.
2. It requires carrying additional metadata around to denote the members which are required to be initialized by the caller.

Further if we believe there is value here in the overall scenario of forcing object creators to be warned / error'd about specific fields then this likely makes sense as a general feature. There is no reason it should be limited to just `init` members.

Resolution There will be no warning on consumption of `init` fields and properties.

LDM wants to have a broader discussion on the idea of required fields and properties. That may cause us to come back and reconsider our position on `init` members and validation.

Allow init as a field modifier

In the same way `init` can serve as a property accessor it could also serve as a designation on fields to give them similar behaviors as `init` properties. That would allow for the field to be assigned before construction was complete by the type, derived types, or object initializers.

```
class Student
{
    public init string FirstName;
    public init string LastName;
}

var s = new Student()
{
    FirstName = "Jarde",
    LastName = "Parsons",
}

s.FirstName = "Jared"; // Error FirstName is readonly
```

In metadata these fields would be marked in the same way as `readonly` fields but with an additional attribute or modreq to indicate they are `init` style fields.

Resolution LDM agrees this proposal is sound but overall the scenario felt disjoint from properties. The decision was to proceed only with `init` properties for now. This has a suitable level of flexibility as an `init` property can mutate a `readonly` field on the declaring type of the property. This will be reconsidered if there is significant customer feedback that justifies the scenario.

Allow init as a type modifier

In the same way the `readonly` modifier can be applied to a `struct` to automatically declare all fields as `readonly`, the `init` only modifier can be declared on a `struct` or `class` to automatically mark all fields as `init`. This means the following two type declarations are equivalent:

```

struct Point
{
    public init int X;
    public init int Y;
}

// vs.

init struct Point
{
    public int X;
    public int Y;
}

```

Resolution This feature is too *cute* here and conflicts with the `readonly struct` feature on which it is based. The `readonly struct` feature is simple in that it applies `readonly` to all members: fields, methods, etc ... The `init struct` feature would only apply to properties. This actually ends up making it confusing for users.

Given that `init` is only valid on certain aspects of a type, we rejected the idea of having it as a type modifier.

Considerations

Compatibility

The `init` feature is designed to be compatible with existing `get` only properties. Specifically it is meant to be a completely additive change for a property which is `get` only today but desires more flexible object creation semantics.

For example consider the following type:

```

class Name
{
    public string First { get; }
    public string Last { get; }

    public Name(string first, string last)
    {
        First = first;
        Last = last;
    }
}

```

Adding `init` to these properties is a non-breaking change:

```

class Name
{
    public string First { get; init; }
    public string Last { get; init; }

    public Name(string first, string last)
    {
        First = first;
        Last = last;
    }
}

```

IL verification

When .NET Core decides to re-implement IL verification, the rules will need to be adjusted to account for `init` members. This will need to be included in the rule changes for non-mutating access to `readonly` data.

The IL verification rules will need to be broken into two parts:

1. Allowing `init` members to set a `readonly` field.
2. Determining when an `init` member can be legally called.

The first is a simple adjustment to the existing rules. The IL verifier can be taught to recognize `init` members and from there it just needs to consider a `readonly` field to be settable on `this` in such a member.

The second rule is more complicated. In the simple case of object initializers the rule is straight forward. It should be legal to call `init` members when the result of a `new` expression is still on the stack. That is until the value has been stored in a local, array element or field or passed as an argument to another method it will still be legal to call `init` members. This ensures that once the result of the `new` expression is published to a named identifier (other than `this`) then it will no longer be legal to call `init` members.

The more complicated case though is when we mix `init` members, object initializers and `await`. That can cause the newly created object to be temporarily hoisted into a state machine and hence put into a field.

```
var student = new Student()
{
    Name = await SomeMethod()
};
```

Here the result of `new Student()` will be hoisted into a state machine as a field before the set of `Name` occurs. The compiler will need to mark such hoisted fields in a way that the IL verifier understands they're not user accessible and hence doesn't violate the intended semantics of `init`.

init members

The `init` modifier could be extended to apply to all instance members. This would generalize the concept of `init` during object construction and allow types to declare helper methods that could participate in the construction process to initialize `init` fields and properties.

Such members would have all the restrictions that an `init` accessor does in this design. The need is questionable though and this can be safely added in a future version of the language in a compatible manner.

Generate three accessors

One potential implementation of `init` properties is to make `init` completely separate from `set`. That means that a property can potentially have three different accessors: `get`, `set`, and `init`.

This has the potential advantage of allowing the use of `modreq` to enforce correctness while maintaining binary compatibility. The implementation would roughly be the following:

1. An `init` accessor is always emitted if there is a `set`. When not defined by the developer it is simply a reference to `set`.
2. The set of a property in an object initializer will always use `init` if present but fall back to `set` if it's missing.

This means that a developer can always safely delete `init` from a property.

The downside of this design is that is only useful if `init` is **always** emitted when there is a `set`. The language can't know if `init` was deleted in the past, it has to assume it was and hence the `init` must always be emitted. That would cause a significant metadata expansion and is simply not worth the cost of the compatibility here.

Target-typed new expressions

12/28/2021 • 3 minutes to read • [Edit Online](#)

Summary

Do not require type specification for constructors when the type is known.

Motivation

Allow field initialization without duplicating the type.

```
Dictionary<string, List<int>> field = new() {  
    { "item1", new() { 1, 2, 3 } }  
};
```

Allow omitting the type when it can be inferred from usage.

```
XmlReader.Create(reader, new() { IgnoreWhitespace = true });
```

Instantiate an object without spelling out the type.

```
private readonly static object s_syncObj = new();
```

Specification

A new syntactic form, *target_typed_new* of the *object_creation_expression* is accepted in which the *type* is optional.

```
object_creation_expression  
    : 'new' type '(' argument_list? ')' object_or_collection_initializer?  
    | 'new' type object_or_collection_initializer  
    | target_typed_new  
    ;  
target_typed_new  
    : 'new' '(' argument_list? ')' object_or_collection_initializer?  
    ;
```

A *target_typed_new* expression does not have a type. However, there is a new *object creation conversion* that is an implicit conversion from expression, that exists from a *target_typed_new* to every type.

Given a target type T , the type T_0 is T 's underlying type if T is an instance of `System.Nullable`. Otherwise T_0 is T . The meaning of a *target_typed_new* expression that is converted to the type T is the same as the meaning of a corresponding *object_creation_expression* that specifies T_0 as the type.

It is a compile-time error if a *target_typed_new* is used as an operand of a unary or binary operator, or if it is used where it is not subject to an *object creation conversion*.

Open Issue: should we allow delegates and tuples as the target-type?

The above rules include delegates (a reference type) and tuples (a struct type). Although both types are

constructible, if the type is inferable, an anonymous function or a tuple literal can already be used.

```
(int a, int b) t = new(1, 2); // "new" is redundant
Action a = new(() => {}); // "new" is redundant

(int a, int b) t = new(); // OK; same as (0, 0)
Action a = new(); // no constructor found
```

Miscellaneous

The following are consequences of the specification:

- `throw new()` is allowed (the target type is `System.Exception`)
- Target-typed `new` is not allowed with binary operators.
- It is disallowed when there is no type to target: unary operators, collection of a `foreach`, in a `using`, in a deconstruction, in an `await` expression, as an anonymous type property (`new { Prop = new() }`), in a `lock` statement, in a `sizeof`, in a `fixed` statement, in a member access (`new().field`), in a dynamically dispatched operation (`someDynamic.Method(new())`), in a LINQ query, as the operand of the `is` operator, as the left operand of the `??` operator, ...
- It is also disallowed as a `ref`.
- The following kinds of types are not permitted as targets of the conversion
 - **Enum types:** `new()` will work (as `new Enum()` works to give the default value), but `new(1)` will not work as enum types do not have a constructor.
 - **Interface types:** This would work the same as the corresponding creation expression for COM types.
 - **Array types:** arrays need a special syntax to provide the length.
 - **dynamic:** we don't allow `new dynamic()`, so we don't allow `new()` with `dynamic` as a target type.
 - **tuples:** These have the same meaning as an object creation using the underlying type.
 - All the other types that are not permitted in the *object_creation_expression* are excluded as well, for instance, pointer types.

Drawbacks

There were some concerns with target-typed `new` creating new categories of breaking changes, but we already have that with `null` and `default`, and that has not been a significant problem.

Alternatives

Most of complaints about types being too long to duplicate in field initialization is about *type arguments* not the type itself, we could infer only type arguments like `new Dictionary(...)` (or similar) and infer type arguments locally from arguments or the collection initializer.

Questions

- Should we forbid usages in expression trees? (no)
- How the feature interacts with `dynamic` arguments? (no special treatment)
- How IntelliSense should work with `new()`? (only when there is a single target-type)

Design meetings

- [LDM-2017-10-18](#)
- [LDM-2018-05-21](#)
- [LDM-2018-06-25](#)

- [LDM-2018-08-22](#)
- [LDM-2018-10-17](#)
- [LDM-2020-03-25](#)

Module Initializers

12/28/2021 • 2 minutes to read • [Edit Online](#)

Summary

Although the .NET platform has a [feature](#) that directly supports writing initialization code for the assembly (technically, the module), it is not exposed in C#. This is a rather niche scenario, but once you run into it the solutions appear to be pretty painful. There are reports of [a number of customers](#) (inside and outside Microsoft) struggling with the problem, and there are no doubt more undocumented cases.

Motivation

- Enable libraries to do eager, one-time initialization when loaded, with minimal overhead and without the user needing to explicitly call anything
- One particular pain point of current `static` constructor approaches is that the runtime must do additional checks on usage of a type with a static constructor, in order to decide whether the static constructor needs to be run or not. This adds measurable overhead.
- Enable source generators to run some global initialization logic without the user needing to explicitly call anything

Detailed design

A method can be designated as a module initializer by decorating it with a `[ModuleInitializer]` attribute.

```
using System;
namespace System.Runtime.CompilerServices
{
    [AttributeUsage(AttributeTargets.Method, AllowMultiple = false)]
    public sealed class ModuleInitializerAttribute : Attribute { }
}
```

The attribute can be used like this:

```
using System.Runtime.CompilerServices;
class C
{
    [ModuleInitializer]
    internal static void M1()
    {
        // ...
    }
}
```

Some requirements are imposed on the method targeted with this attribute:

1. The method must be `static`.
2. The method must be parameterless.
3. The method must return `void`.
4. The method must not be generic or be contained in a generic type.
5. The method must be accessible from the containing module.
 - This means the method's effective accessibility must be `internal` or `public`.

- This also means the method cannot be a local function.

When one or more valid methods with this attribute are found in a compilation, the compiler will emit a module initializer which calls each of the attributed methods. The calls will be emitted in a reserved, but deterministic order.

Drawbacks

Why should we *not* do this?

- Perhaps the existing third-party tooling for "injecting" module initializers is sufficient for users who have been asking for this feature.

Design meetings

[April 8th, 2020](#)

Extending Partial Methods

12/28/2021 • 5 minutes to read • [Edit Online](#)

Summary

This proposal aims to remove all restrictions around the signatures of `partial` methods in C#. The goal being to expand the set of scenarios in which these methods can work with source generators as well as being a more general declaration form for C# methods.

See also the [original partial methods specification](#).

Motivation

C# has limited support for developers splitting methods into declarations and definitions / implementations.

```
partial class C
{
    // The declaration of C.M
    partial void M(string message);
}

partial class C
{
    // The definition of C.M
    partial void M(string message) => Console.WriteLine(message);
}
```

One behavior of `partial` methods is that when the definition is absent then the language will simply erase any calls to the `partial` method. Essentially it behaves like a call to a `[Conditional]` method where the condition was evaluated to false.

```
partial class D
{
    partial void M(string message);

    void Example()
    {
        M(GetIt()); // Call to M and GetIt erased at compile time
    }

    string GetIt() => "Hello World";
}
```

The original motivation for this feature was source generation in the form of designer generated code. Users were constantly editing the generated code because they wanted to hook some aspect of the generated code. Most notably parts of the Windows Forms startup process, after components were initialized.

Editing the generated code was error prone because any action which caused the designer to regenerate the code would cause the user edit to be erased. The `partial` method feature eased this tension because it allowed designers to emit hooks in the form of `partial` methods.

Designers could emit hooks like `partial void OnComponentInit()` and developers could define declarations for them or not define them. In either case though the generated code would compile and developers who were interested in the process could hook in as needed.

This does mean that partial methods have several restrictions:

1. Must have a `void` return type.
2. Cannot have `out` parameters.
3. Cannot have any accessibility (implicitly `private`).

These restrictions exist because the language must be able to emit code when the call site is erased. Given they can be erased `private` is the only possible accessibility because the member can't be exposed in assembly metadata. These restrictions also serve to limit the set of scenarios in which `partial` methods can be applied.

The proposal here is to remove all of the existing restrictions around `partial` methods. Essentially let them have `out`, non-void return types or any type of accessibility. Such `partial` declarations would then have the added requirement that a definition must exist. That means the language does not have to consider the impact of erasing the call sites.

This would expand the set of generator scenarios that `partial` methods could participate in and hence link in nicely with our source generators feature. For example a regex could be defined using the following pattern:

```
[RegexGenerated("(dog|cat|fish)")]  
partial bool IsPetMatch(string input);
```

This gives both the developer a simple declarative way of opting into generators as well as giving generators a very easy set of declarations to look through in the source code to drive their generated output.

Compare that with the difficulty that a generator would have hooking up the following snippet of code.

```
var regex = new RegularExpression("(dog|cat|fish)");  
if (regex.IsMatch(someInput))  
{  
  
}
```

Given that the compiler doesn't allow generators to modify code hooking up this pattern would be pretty much impossible for generators. They would need to resort to reflection in the `IsMatch` implementation, or asking users to change their call sites to a new method + refactor the regex to pass the string literal as an argument. It's pretty messy.

Detailed Design

The language will change to allow `partial` methods to be annotated with an explicit accessibility modifier. This means they can be labeled as `private`, `public`, etc ...

When a `partial` method has an explicit accessibility modifier though the language will require that the declaration has a matching definition even when the accessibility is `private`:

```

partial class C
{
    // Okay because no definition is required here
    partial void M1();

    // Okay because M2 has a definition
    private partial void M2();

    // Error: partial method M3 must have a definition
    private partial void M3();
}

partial class C
{
    private partial void M2() { }
}

```

Further the language will remove all restrictions on what can appear on a `partial` method which has an explicit accessibility. Such declarations can contain non-void return types, `out` parameters, `extern` modifier, etc ... These signatures will have the full expressivity of the C# language.

```

partial class D
{
    // Okay
    internal partial bool TryParse(string s, out int i);
}

partial class D
{
    internal partial bool TryParse(string s, out int i) { }
}

```

This explicitly allows for `partial` methods to participate in `overrides` and `interface` implementations:

```

interface IStudent
{
    string GetName();
}

partial class C : IStudent
{
    public virtual partial string GetName();
}

partial class C
{
    public virtual partial string GetName() => "Jarde";
}

```

The compiler will change the error it emits when a `partial` method contains an illegal element to essentially say:

Cannot use `ref` on a `partial` method that lacks explicit accessibility

This will help point developers in the right direction when using this feature.

Restrictions:

- `partial` declarations with explicit accessibility must have a definition
- `partial` declarations and definition signatures must match on all method and parameter modifiers. The only

aspects which can differ are parameter names and attribute lists (this is not new but rather an existing requirement of `partial` methods).

Questions

partial on all members

Given that we're expanding `partial` to be more friendly to source generators should we also expand it to work on all class members? For example should we be able to declare `partial` constructors, operators, etc ...

Resolution The idea is sound but at this point in the C# 9 schedule we're trying to avoid unnecessary feature creep. Want to solve the immediate problem of expanding the feature to work with modern source generators.

Extending `partial` to support other members will be considered for the C# 10 release. Seems likely that we will consider this extension.

Use abstract instead of partial

The crux of this proposal is essentially ensuring that a declaration has a corresponding definition / implementation. Given that should we use `abstract` since it's already a language keyword that forces the developer to think about having an implementation?

Resolution There was a healthy discussion about this but eventually it was decided against. Yes the requirements are familiar but the concepts are significantly different. Could easily lead the developer to believe they were creating virtual slots when they were not doing so.

Static anonymous functions

12/28/2021 • 2 minutes to read • [Edit Online](#)

Summary

Allow a 'static' modifier on lambdas and anonymous methods, which disallows capture of locals or instance state from containing scopes.

Motivation

Avoid unintentionally capturing state from the enclosing context, which can result in unexpected retention of captured objects or unexpected additional allocations.

Detailed design

A lambda or anonymous method may have a `static` modifier. The `static` modifier indicates that the lambda or anonymous method is a *static anonymous function*.

A *static anonymous function* cannot capture state from the enclosing scope. As a result, locals, parameters, and `this` from the enclosing scope are not available within a *static anonymous function*.

A *static anonymous function* cannot reference instance members from an implicit or explicit `this` or `base` reference.

A *static anonymous function* may reference `static` members from the enclosing scope.

A *static anonymous function* may reference `constant` definitions from the enclosing scope.

`nameof()` in a *static anonymous function* may reference locals, parameters, or `this` or `base` from the enclosing scope.

Accessibility rules for `private` members in the enclosing scope are the same for `static` and non-`static` anonymous functions.

No guarantee is made as to whether a *static anonymous function* definition is emitted as a `static` method in metadata. This is left up to the compiler implementation to optimize.

A non-`static` local function or anonymous function can capture state from an enclosing *static anonymous function* but cannot capture state outside the enclosing *static anonymous function*.

Removing the `static` modifier from an anonymous function in a valid program does not change the meaning of the program.

Target-Typed Conditional Expression

12/28/2021 • 3 minutes to read • [Edit Online](#)

Conditional Expression Conversion

For a conditional expression `c ? e1 : e2`, when

1. there is no common type for `e1` and `e2`, or
2. for which a common type exists but one of the expressions `e1` or `e2` has no implicit conversion to that type

we define a new implicit *conditional expression conversion* that permits an implicit conversion from the conditional expression to any type `T` for which there is a conversion-from-expression from `e1` to `T` and also from `e2` to `T`. It is an error if a conditional expression neither has a common type between `e1` and `e2` nor is subject to a *conditional expression conversion*.

Better Conversion from Expression

We change

Better conversion from expression

Given an implicit conversion `c1` that converts from an expression `E` to a type `T1`, and an implicit conversion `c2` that converts from an expression `E` to a type `T2`, `c1` is a **better conversion** than `c2` if `E` does not exactly match `T2` and at least one of the following holds:

- `E` exactly matches `T1` ([Exactly matching Expression](#))
- `T1` is a better conversion target than `T2` ([Better conversion target](#))

to

Better conversion from expression

Given an implicit conversion `c1` that converts from an expression `E` to a type `T1`, and an implicit conversion `c2` that converts from an expression `E` to a type `T2`, `c1` is a **better conversion** than `c2` if `E` does not exactly match `T2` and at least one of the following holds:

- `E` exactly matches `T1` ([Exactly matching Expression](#))
- `c1` is not a *conditional expression conversion* and `c2` is a *conditional expression conversion*.
- `T1` is a better conversion target than `T2` ([Better conversion target](#)) and either `c1` and `c2` are both *conditional expression conversions* or neither is a *conditional expression conversion*.

Cast Expression

The current C# language specification says

A *cast_expression* of the form `(T)E`, where `T` is a *type* and `E` is a *unary_expression*, performs an explicit conversion ([Explicit conversions](#)) of the value of `E` to type `T`.

In the presence of the *conditional expression conversion* there may be more than one possible conversion from `E` to `T`. With the addition of *conditional expression conversion*, we prefer any other conversion to a *conditional expression conversion*, and use the *conditional expression conversion* only as a last resort.

Design Notes

The reason for the change to *Better conversion from expression* is to handle a case such as this:

```
M(b ? 1 : 2);

void M(short);
void M(long);
```

This approach does have two small downsides. First, it is not quite the same as the switch expression:

```
M(b ? 1 : 2); // calls M(long)
M(b switch { true => 1, false => 2 }); // calls M(short)
```

This is still a breaking change, but its scope is less likely to affect real programs:

```
M(b ? 1 : 2, 1); // calls M(long, long) without this feature; ambiguous with this feature.

M(short, short);
M(long, long);
```

This becomes ambiguous because the conversion to `long` is better for the first argument (because it does not use the *conditional expression conversion*), but the conversion to `short` is better for the second argument (because `short` is a *better conversion target* than `long`). This breaking change seems less serious because it does not silently change the behavior of an existing program.

The reason for the notes on the cast expression is to handle a case such as this:

```
_ = (short)(b ? 1 : 2);
```

This program currently uses the explicit conversion from `int` to `short`, and we want to preserve the current language meaning of this program. The change would be unobservable at runtime, but with the following program the change would be observable:

```
_ = (A)(b ? c : d);
```

where `c` is of type `C`, `d` is of type `D`, and there is an implicit user-defined conversion from `C` to `D`, and an implicit user-defined conversion from `D` to `A`, and an implicit user-defined conversion from `C` to `A`. If this code is compiled before C# 9.0, when `b` is true we convert from `C` to `D` then to `A`. If we use the *conditional expression conversion*, then when `b` is true we convert from `C` to `A` directly, which executes a different sequence of user code. Therefore we treat the *conditional expression conversion* as a last resort in a cast, to preserve existing behavior.

Covariant returns

12/28/2021 • 10 minutes to read • [Edit Online](#)

Summary

Support *covariant return types*. Specifically, permit the override of a method to declare a more derived return type than the method it overrides, and similarly to permit the override of a read-only property to declare a more derived type. Override declarations appearing in more derived types would be required to provide a return type at least as specific as that appearing in overrides in its base types. Callers of the method or property would statically receive the more refined return type from an invocation.

Motivation

It is a common pattern in code that different method names have to be invented to work around the language constraint that overrides must return the same type as the overridden method.

This would be useful in the factory pattern. For example, in the Roslyn code base we would have

```
class Compilation ...
{
    public virtual Compilation WithOptions(Options options)...
}
```

```
class CSharpCompilation : Compilation
{
    public override CSharpCompilation WithOptions(Options options)...
}
```

Detailed design

This is a specification for [covariant return types](#) in C#. Our intent is to permit the override of a method to return a more derived return type than the method it overrides, and similarly to permit the override of a read-only property to return a more derived return type. Callers of the method or property would statically receive the more refined return type from an invocation, and overrides appearing in more derived types would be required to provide a return type at least as specific as that appearing in overrides in its base types.

Class Method Override

The [existing constraint on class override](#) methods

- The override method and the overridden base method have the same return type.

is modified to

- The override method must have a return type that is convertible by an identity conversion or (if the method has a value return - not a [ref return](#)) implicit reference conversion to the return type of the overridden base method.

And the following additional requirements are appended to that list:

- The override method must have a return type that is convertible by an identity conversion or (if the method has a value return - not a [ref return](#)) implicit reference conversion to the return type of every override of the overridden base method that is declared in a (direct or indirect) base type of the override method.
- The override method's return type must be at least as accessible as the override method ([Accessibility domains](#)).

This constraint permits an override method in a `private` class to have a `private` return type. However it requires a `public` override method in a `public` type to have a `public` return type.

Class Property and Indexer Override

The [existing constraint on class override](#) properties

An overriding property declaration shall specify the exact same accessibility modifiers and name as the inherited property, and there shall be an identity conversion ~~between the type of the overriding and the inherited property~~. If the inherited property has only a single accessor (i.e., if the inherited property is read-only or write-only), the overriding property shall include only that accessor. If the inherited property includes both accessors (i.e., if the inherited property is read-write), the overriding property can include either a single accessor or both accessors.

is modified to

An overriding property declaration shall specify the exact same accessibility modifiers and name as the inherited property, and there shall be an identity conversion **or (if the inherited property is read-only and has a value return - not a [ref return](#)) implicit reference conversion from the type of the overriding property to the type of the inherited property**. If the inherited property has only a single accessor (i.e., if the inherited property is read-only or write-only), the overriding property shall include only that accessor. If the inherited property includes both accessors (i.e., if the inherited property is read-write), the overriding property can include either a single accessor or both accessors. **The overriding property's type must be at least as accessible as the overriding property ([Accessibility domains](#)).**

The remainder of the draft specification below proposes a further extension to covariant returns of interface methods to be considered later.

Interface Method, Property, and Indexer Override

Adding to the kinds of members that are permitted in an interface with the addition of the DIM feature in C# 8.0, we further add support for `override` members along with covariant returns. These follow the rules of `override` members as specified for classes, with the following differences:

The following text in classes:

The method overridden by an override declaration is known as the ***overridden base method***. For an override method `M` declared in a class `C`, the overridden base method is determined by examining each base class of `C`, starting with the direct base class of `C` and continuing with each successive direct base class, until in a given base class type at least one accessible method is located which has the same signature as `M` after substitution of type arguments.

is given the corresponding specification for interfaces:

The method overridden by an override declaration is known as the ***overridden base method***. For an override method `M` declared in an interface `I`, the overridden base method is determined by examining each direct or indirect base interface of `I`, collecting the set of interfaces declaring an accessible method

which has the same signature as `M` after substitution of type arguments. If this set of interfaces has a *most derived type*, to which there is an identity or implicit reference conversion from every type in this set, and that type contains a unique such method declaration, then that is the *overridden base method*.

We similarly permit `override` properties and indexers in interfaces as specified for classes in *15.7.6 Virtual, sealed, override, and abstract accessors*.

Name Lookup

Name lookup in the presence of class `override` declarations currently modify the result of name lookup by imposing on the found member details from the most derived `override` declaration in the class hierarchy starting from the type of the identifier's qualifier (or `this` when there is no qualifier). For example, in *12.6.2.2 Corresponding parameters* we have

For virtual methods and indexers defined in classes, the parameter list is picked from the first declaration or override of the function member found when starting with the static type of the receiver, and searching through its base classes.

to this we add

For virtual methods and indexers defined in interfaces, the parameter list is picked from the declaration or override of the function member found in the most derived type among those types containing the declaration of override of the function member. It is a compile-time error if no unique such type exists.

For the result type of a property or indexer access, the existing text

- If `I` identifies an instance property, then the result is a property access with an associated instance expression of `E` and an associated type that is the type of the property. If `T` is a class type, the associated type is picked from the first declaration or override of the property found when starting with `T`, and searching through its base classes.

is augmented with

If `T` is an interface type, the associated type is picked from the declaration or override of the property found in the most derived of `T` or its direct or indirect base interfaces. It is a compile-time error if no unique such type exists.

A similar change should be made in *12.7.7.3 Indexer access*

In *12.7.6 Invocation expressions* we augment the existing text

- Otherwise, the result is a value, with an associated type of the return type of the method or delegate. If the invocation is of an instance method, and the receiver is of a class type `T`, the associated type is picked from the first declaration or override of the method found when starting with `T` and searching through its base classes.

with

If the invocation is of an instance method, and the receiver is of an interface type `T`, the associated type is picked from the declaration or override of the method found in the most derived interface from among `T` and its direct and indirect base interfaces. It is a compile-time error if no unique such type exists.

Implicit Interface Implementations

This section of the specification

For purposes of interface mapping, a class member `A` matches an interface member `B` when:

- `A` and `B` are methods, and the name, type, and formal parameter lists of `A` and `B` are identical.
- `A` and `B` are properties, the name and type of `A` and `B` are identical, and `A` has the same accessors as `B` (`A` is permitted to have additional accessors if it is not an explicit interface member implementation).
- `A` and `B` are events, and the name and type of `A` and `B` are identical.
- `A` and `B` are indexers, the type and formal parameter lists of `A` and `B` are identical, and `A` has the same accessors as `B` (`A` is permitted to have additional accessors if it is not an explicit interface member implementation).

is modified as follows:

For purposes of interface mapping, a class member `A` matches an interface member `B` when:

- `A` and `B` are methods, and the name and formal parameter lists of `A` and `B` are identical, and the return type of `A` is convertible to the return type of `B` via an identity of implicit reference conversion to the return type of `B`.
- `A` and `B` are properties, the name of `A` and `B` are identical, `A` has the same accessors as `B` (`A` is permitted to have additional accessors if it is not an explicit interface member implementation), and the type of `A` is convertible to the return type of `B` via an identity conversion or, if `A` is a readonly property, an implicit reference conversion.
- `A` and `B` are events, and the name and type of `A` and `B` are identical.
- `A` and `B` are indexers, the formal parameter lists of `A` and `B` are identical, `A` has the same accessors as `B` (`A` is permitted to have additional accessors if it is not an explicit interface member implementation), and the type of `A` is convertible to the return type of `B` via an identity conversion or, if `A` is a readonly indexer, an implicit reference conversion.

This is technically a breaking change, as the program below prints "C1.M" today, but would print "C2.M" under the proposed revision.

```
using System;

interface I1 { object M(); }
class C1 : I1 { public object M() { return "C1.M"; } }
class C2 : C1, I1 { public new string M() { return "C2.M"; } }
class Program
{
    static void Main()
    {
        I1 i = new C2();
        Console.WriteLine(i.M());
    }
}
```

Due to this breaking change, we might consider not supporting covariant return types on implicit implementations.

Constraints on Interface Implementation

We will need a rule that an explicit interface implementation must declare a return type no less derived than the return type declared in any override in its base interfaces.

API Compatibility Implications

TBD

Open Issues

The specification does not say how the caller gets the more refined return type. Presumably that would be done in a way similar to the way that callers get the most derived override's parameter specifications.

If we have the following interfaces:

```
interface I1 { I1 M(); }
interface I2 { I2 M(); }
interface I3: I1, I2 { override I3 M(); }
```

Note that in `I3`, the methods `I1.M()` and `I2.M()` have been "merged". When implementing `I3`, it is necessary to implement them both together.

Generally, we require an explicit implementation to refer to the original method. The question is, in a class

```
class C : I1, I2, I3
{
    C IN.M();
}
```

What does that mean here? What should *N* be?

I suggest that we permit implementing either `I1.M` or `I2.M` (but not both), and treat that as an implementation of both.

Drawbacks

- [] Every language change must pay for itself.
- [] We should ensure that the performance is reasonable, even in the case of deep inheritance hierarchies
- [] We should ensure that artifacts of the translation strategy do not affect language semantics, even when consuming new IL from old compilers.

Alternatives

We could relax the language rules slightly to allow, in source,

```
abstract class Cloneable
{
    public abstract Cloneable Clone();
}

class Digit : Cloneable
{
    public override Cloneable Clone()
    {
        return this.Clone();
    }

    public new Digit Clone() // Error: 'Digit' already defines a member called 'Clone' with the same
parameter types
    {
        return this;
    }
}
```

Unresolved questions

- [] How will APIs that have been compiled to use this feature work in older versions of the language?

Design meetings

- some discussion at <https://github.com/dotnet/roslyn/issues/357>.
- <https://github.com/dotnet/csharplang/blob/master/meetings/2020/LDM-2020-01-08.md>
- Offline discussion toward a decision to support overriding of class methods only in C# 9.0.

Extension `GetEnumerator` support for `foreach` loops.

12/28/2021 • 5 minutes to read • [Edit Online](#)

Summary

Allow `foreach` loops to recognize an extension method `GetEnumerator` method that otherwise satisfies the `foreach` pattern, and loop over the expression when it would otherwise be an error.

Motivation

This will bring `foreach` inline with how other features in C# are implemented, including `async` and pattern-based deconstruction.

Detailed design

The spec change is relatively straightforward. We modify `The foreach statement` section to this text:

The compile-time processing of a `foreach` statement first determines the *collection type*, *enumerator type* and *element type* of the expression. This determination proceeds as follows:

- If the type `x` of *expression* is an array type then there is an implicit reference conversion from `x` to the `IEnumerable` interface (since `System.Array` implements this interface). The *collection type* is the `IEnumerable` interface, the *enumerator type* is the `IEnumerator` interface and the *element type* is the element type of the array type `x`.
- If the type `x` of *expression* is `dynamic` then there is an implicit conversion from *expression* to the `IEnumerable` interface ([Implicit dynamic conversions](#)). The *collection type* is the `IEnumerable` interface and the *enumerator type* is the `IEnumerator` interface. If the `var` identifier is given as the *local_variable_type* then the *element type* is `dynamic`, otherwise it is `object`.
- Otherwise, determine whether the type `x` has an appropriate `GetEnumerator` method:
 - Perform member lookup on the type `x` with identifier `GetEnumerator` and no type arguments. If the member lookup does not produce a match, or it produces an ambiguity, or produces a match that is not a method group, check for an enumerable interface as described below. It is recommended that a warning be issued if member lookup produces anything except a method group or no match.
 - Perform overload resolution using the resulting method group and an empty argument list. If overload resolution results in no applicable methods, results in an ambiguity, or results in a single best method but that method is either static or not public, check for an enumerable interface as described below. It is recommended that a warning be issued if overload resolution produces anything except an unambiguous public instance method or no applicable methods.
 - If the return type `E` of the `GetEnumerator` method is not a class, struct or interface type, an error is produced and no further steps are taken.
 - Member lookup is performed on `E` with the identifier `Current` and no type arguments. If the member lookup produces no match, the result is an error, or the result is anything except a public instance property that permits reading, an error is produced and no further steps are taken.
 - Member lookup is performed on `E` with the identifier `MoveNext` and no type arguments. If the

member lookup produces no match, the result is an error, or the result is anything except a method group, an error is produced and no further steps are taken.

- Overload resolution is performed on the method group with an empty argument list. If overload resolution results in no applicable methods, results in an ambiguity, or results in a single best method but that method is either static or not public, or its return type is not `bool`, an error is produced and no further steps are taken.

- The **collection type** is `X`, the **enumerator type** is `E`, and the **element type** is the type of the `Current` property.

- Otherwise, check for an enumerable interface:

- If among all the types `Ti` for which there is an implicit conversion from `X` to `IEnumerable<Ti>`, there is a unique type `T` such that `T` is not `dynamic` and for all the other `Ti` there is an implicit conversion from `IEnumerable<T>` to `IEnumerable<Ti>`, then the **collection type** is the interface `IEnumerable<T>`, the **enumerator type** is the interface `IEnumerator<T>`, and the **element type** is `T`.

- Otherwise, if there is more than one such type `T`, then an error is produced and no further steps are taken.

- Otherwise, if there is an implicit conversion from `X` to the `System.Collections.IEnumerable` interface, then the **collection type** is this interface, the **enumerator type** is the interface `System.Collections.IEnumerator`, and the **element type** is `object`.

- Otherwise, determine whether the type 'X' has an appropriate `GetEnumerator` extension method:

- Perform extension method lookup on the type `X` with identifier `GetEnumerator`. If the member lookup does not produce a match, or it produces an ambiguity, or produces a match which is not a method group, an error is produced and no further steps are taken. It is recommended that a warning be issues if member lookup produces anything except a method group or no match.

- Perform overload resolution using the resulting method group and a single argument of type `X`. If overload resolution produces no applicable methods, results in an ambiguity, or results in a single best method but that method is not accessible, an error is produced and no further steps are taken.

- This resolution permits the first argument to be passed by ref if `X` is a struct type, and the ref kind is `in`.

- If the return type `E` of the `GetEnumerator` method is not a class, struct or interface type, an error is produced and no further steps are taken.

- Member lookup is performed on `E` with the identifier `Current` and no type arguments. If the member lookup produces no match, the result is an error, or the result is anything except a public instance property that permits reading, an error is produced and no further steps are taken.

- Member lookup is performed on `E` with the identifier `MoveNext` and no type arguments. If the member lookup produces no match, the result is an error, or the result is anything except a method group, an error is produced and no further steps are taken.

- Overload resolution is performed on the method group with an empty argument list. If overload resolution results in no applicable methods, results in an ambiguity, or results in a single best method but that method is either static or not public, or its return type is not `bool`, an error is produced and no further steps are taken.

- The **collection type** is `X`, the **enumerator type** is `E`, and the **element type** is the type of the `Current` property.

- Otherwise, an error is produced and no further steps are taken.

For `await foreach`, the rules are similarly modified. The only change that is required to that spec is removing the `Extension methods do not contribute.` line from the description, as the rest of that spec is based on the

above rules with different names substituted for the pattern methods.

Drawbacks

Every change adds additional complexity to the language, and this potentially allows things that weren't designed to be `foreach` ed to be `foreach` ed, like `Range` .

Alternatives

Doing nothing.

Unresolved questions

None at this point.

Lambda discard parameters

12/28/2021 • 2 minutes to read • [Edit Online](#)

Summary

Allow discards (`_`) to be used as parameters of lambdas and anonymous methods. For example:

- lambdas: `(_, _) => 0`, `(int _, int _) => 0`
- anonymous methods: `delegate(int _, int _) { return 0; }`

Motivation

Unused parameters do not need to be named. The intent of discards is clear, i.e. they are unused/discarded.

Detailed design

Method parameters In the parameter list of a lambda or anonymous method with more than one parameter named `_`, such parameters are discard parameters. Note: if a single parameter is named `_` then it is a regular parameter for backwards compatibility reasons.

Discard parameters do not introduce any names to any scopes. Note this implies they do not cause any `_` (underscore) names to be hidden.

Simple names If `k` is zero and the *simple_name* appears within a *block* and if the *block's* (or an enclosing *block's*) local variable declaration space ([Declarations](#)) contains a local variable, parameter (with the exception of discard parameters) or constant with name `I`, then the *simple_name* refers to that local variable, parameter or constant and is classified as a variable or value.

Scopes With the exception of discard parameters, the scope of a parameter declared in a *lambda_expression* ([Anonymous function expressions](#)) is the *anonymous_function_body* of that *lambda_expression*. With the exception of discard parameters, the scope of a parameter declared in an *anonymous_method_expression* ([Anonymous function expressions](#)) is the *block* of that *anonymous_method_expression*.

Related spec sections

- [Corresponding parameters](#)

Attributes on local functions

12/28/2021 • 2 minutes to read • [Edit Online](#)

Attributes

Local function declarations are now permitted to have [attributes](#). Parameters and type parameters on local functions are also allowed to have attributes.

Attributes with a specified meaning when applied to a method, its parameters, or its type parameters will have the same meaning when applied to a local function, its parameters, or its type parameters, respectively.

A local function can be made conditional in the same sense as a [conditional method](#) by decorating it with a `[ConditionalAttribute]`. A conditional local function must also be `static`. All restrictions on conditional methods also apply to conditional local functions, including that the return type must be `void`.

Extern

The `extern` modifier is now permitted on local functions. This makes the local function external in the same sense as an [external method](#).

Similarly to an external method, the *local-function-body* of an external local function must be a semicolon. A semicolon *local-function-body* is only permitted on an external local function.

An external local function must also be `static`.

Syntax

The [local functions grammar](#) is modified as follows:

```
local-function-header
    : attributes? local-function-modifiers? return-type identifier type-parameter-list?
      ( formal-parameter-list? ) type-parameter-constraints-clauses
    ;

local-function-modifiers
    : (async | unsafe | static | extern)*
    ;

local-function-body
    : block
    | arrow-expression-body
    | ';'
    ;
```

Native-sized integers

12/28/2021 • 10 minutes to read • [Edit Online](#)

Summary

Language support for a native-sized signed and unsigned integer types.

The motivation is for interop scenarios and for low-level libraries.

Design

The identifiers `nint` and `nuint` are new contextual keywords that represent native signed and unsigned integer types. The identifiers are only treated as keywords when name lookup does not find a viable result at that program location.

```
nint x = 3;
string y = nameof(nuint);
_ = nint.Equals(x, 3);
```

The types `nint` and `nuint` are represented by the underlying types `System.IntPtr` and `System.UIntPtr` with compiler surfacing additional conversions and operations for those types as native ints.

Constants

Constant expressions may be of type `nint` or `nuint`. There is no direct syntax for native int literals. Implicit or explicit casts of other integral constant values can be used instead: `const nint i = (nint)42;`

`nint` constants are in the range [`int.MinValue`, `int.MaxValue`].

`nuint` constants are in the range [`uint.MinValue`, `uint.MaxValue`].

There are no `MinValue` or `MaxValue` fields on `nint` or `nuint` because, other than `nuint.MinValue`, those values cannot be emitted as constants.

Constant folding is supported for all unary operators { `+`, `-`, `~` } and binary operators { `+`, `-`, `*`, `/`, `%`, `==`, `!=`, `<`, `<=`, `>`, `>=`, `&`, `|`, `^`, `<<`, `>>` }. Constant folding operations are evaluated with `Int32` and `UInt32` operands rather than native ints for consistent behavior regardless of compiler platform. If the operation results in a constant value in 32-bits, constant folding is performed at compile-time. Otherwise the operation is executed at runtime and not considered a constant.

Conversions

There is an identity conversion between `nint` and `IntPtr`, and between `nuint` and `UIntPtr`. There is an identity conversion between compound types that differ by native ints and underlying types only: arrays, `Nullable<>`, constructed types, and tuples.

The tables below cover the conversions between special types. (The IL for each conversion includes the variants for `unchecked` and `checked` contexts if different.)

OPERAND	TARGET	CONVERSION	IL
<code>object</code>	<code>nint</code>	Unboxing	<code>unbox</code>

OPERAND	TARGET	CONVERSION	IL
<code>void*</code>	<code>nint</code>	PointerToVoid	<code>conv.i</code>
<code>sbyte</code>	<code>nint</code>	ImplicitNumeric	<code>conv.i</code>
<code>byte</code>	<code>nint</code>	ImplicitNumeric	<code>conv.u</code>
<code>short</code>	<code>nint</code>	ImplicitNumeric	<code>conv.i</code>
<code>ushort</code>	<code>nint</code>	ImplicitNumeric	<code>conv.u</code>
<code>int</code>	<code>nint</code>	ImplicitNumeric	<code>conv.i</code>
<code>uint</code>	<code>nint</code>	ExplicitNumeric	<code>conv.u</code> / <code>conv.ovf.u</code>
<code>long</code>	<code>nint</code>	ExplicitNumeric	<code>conv.i</code> / <code>conv.ovf.i</code>
<code>ulong</code>	<code>nint</code>	ExplicitNumeric	<code>conv.i</code> / <code>conv.ovf.i</code>
<code>char</code>	<code>nint</code>	ImplicitNumeric	<code>conv.i</code>
<code>float</code>	<code>nint</code>	ExplicitNumeric	<code>conv.i</code> / <code>conv.ovf.i</code>
<code>double</code>	<code>nint</code>	ExplicitNumeric	<code>conv.i</code> / <code>conv.ovf.i</code>
<code>decimal</code>	<code>nint</code>	ExplicitNumeric	<code>long</code> <code>decimal.op_Explicit(decimal)</code> <code>conv.i</code> / <code>... conv.ovf.i</code>
<code>IntPtr</code>	<code>nint</code>	Identity	
<code>UIntPtr</code>	<code>nint</code>	None	
<code>object</code>	<code>nuint</code>	Unboxing	<code>unbox</code>
<code>void*</code>	<code>nuint</code>	PointerToVoid	<code>conv.u</code>
<code>sbyte</code>	<code>nuint</code>	ExplicitNumeric	<code>conv.u</code> / <code>conv.ovf.u</code>
<code>byte</code>	<code>nuint</code>	ImplicitNumeric	<code>conv.u</code>
<code>short</code>	<code>nuint</code>	ExplicitNumeric	<code>conv.u</code> / <code>conv.ovf.u</code>
<code>ushort</code>	<code>nuint</code>	ImplicitNumeric	<code>conv.u</code>
<code>int</code>	<code>nuint</code>	ExplicitNumeric	<code>conv.u</code> / <code>conv.ovf.u</code>
<code>uint</code>	<code>nuint</code>	ImplicitNumeric	<code>conv.u</code>

OPERAND	TARGET	CONVERSION	IL
<code>long</code>	<code>nuint</code>	ExplicitNumeric	<code>conv.u</code> / <code>conv.ovf.u</code>
<code>ulong</code>	<code>nuint</code>	ExplicitNumeric	<code>conv.u</code> / <code>conv.ovf.u</code>
<code>char</code>	<code>nuint</code>	ImplicitNumeric	<code>conv.u</code>
<code>float</code>	<code>nuint</code>	ExplicitNumeric	<code>conv.u</code> / <code>conv.ovf.u</code>
<code>double</code>	<code>nuint</code>	ExplicitNumeric	<code>conv.u</code> / <code>conv.ovf.u</code>
<code>decimal</code>	<code>nuint</code>	ExplicitNumeric	<code>ulong decimal.op_Explicit(decimal) conv.u / ... conv.ovf.u.un</code>
<code>IntPtr</code>	<code>nuint</code>	None	
<code>UIntPtr</code>	<code>nuint</code>	Identity	
Enumeration	<code>nint</code>	ExplicitEnumeration	
Enumeration	<code>nuint</code>	ExplicitEnumeration	

OPERAND	TARGET	CONVERSION	IL
<code>nint</code>	<code>object</code>	Boxing	<code>box</code>
<code>nint</code>	<code>void*</code>	PointerToVoid	<code>conv.i</code>
<code>nint</code>	<code>nuint</code>	ExplicitNumeric	<code>conv.u</code> / <code>conv.ovf.u</code>
<code>nint</code>	<code>sbyte</code>	ExplicitNumeric	<code>conv.i1</code> / <code>conv.ovf.i1</code>
<code>nint</code>	<code>byte</code>	ExplicitNumeric	<code>conv.u1</code> / <code>conv.ovf.u1</code>
<code>nint</code>	<code>short</code>	ExplicitNumeric	<code>conv.i2</code> / <code>conv.ovf.i2</code>
<code>nint</code>	<code>ushort</code>	ExplicitNumeric	<code>conv.u2</code> / <code>conv.ovf.u2</code>
<code>nint</code>	<code>int</code>	ExplicitNumeric	<code>conv.i4</code> / <code>conv.ovf.i4</code>
<code>nint</code>	<code>uint</code>	ExplicitNumeric	<code>conv.u4</code> / <code>conv.ovf.u4</code>
<code>nint</code>	<code>long</code>	ImplicitNumeric	<code>conv.i8</code> / <code>conv.ovf.i8</code>
<code>nint</code>	<code>ulong</code>	ExplicitNumeric	<code>conv.i8</code> / <code>conv.ovf.i8</code>
<code>nint</code>	<code>char</code>	ExplicitNumeric	<code>conv.u2</code> / <code>conv.ovf.u2</code>

OPERAND	TARGET	CONVERSION	IL
<code>nint</code>	<code>float</code>	ImplicitNumeric	<code>conv.r4</code>
<code>nint</code>	<code>double</code>	ImplicitNumeric	<code>conv.r8</code>
<code>nint</code>	<code>decimal</code>	ImplicitNumeric	<code>conv.i8 decimal</code> <code>decimal.op_implicit(long)</code>
<code>nint</code>	<code>IntPtr</code>	Identity	
<code>nint</code>	<code>UIntPtr</code>	None	
<code>nint</code>	Enumeration	ExplicitEnumeration	
<code>nuint</code>	<code>object</code>	Boxing	<code>box</code>
<code>nuint</code>	<code>void*</code>	PointerToVoid	<code>conv.u</code>
<code>nuint</code>	<code>nint</code>	ExplicitNumeric	<code>conv.i</code> / <code>conv.ovf.i</code>
<code>nuint</code>	<code>sbyte</code>	ExplicitNumeric	<code>conv.i1</code> / <code>conv.ovf.i1</code>
<code>nuint</code>	<code>byte</code>	ExplicitNumeric	<code>conv.u1</code> / <code>conv.ovf.u1</code>
<code>nuint</code>	<code>short</code>	ExplicitNumeric	<code>conv.i2</code> / <code>conv.ovf.i2</code>
<code>nuint</code>	<code>ushort</code>	ExplicitNumeric	<code>conv.u2</code> / <code>conv.ovf.u2</code>
<code>nuint</code>	<code>int</code>	ExplicitNumeric	<code>conv.i4</code> / <code>conv.ovf.i4</code>
<code>nuint</code>	<code>uint</code>	ExplicitNumeric	<code>conv.u4</code> / <code>conv.ovf.u4</code>
<code>nuint</code>	<code>long</code>	ExplicitNumeric	<code>conv.i8</code> / <code>conv.ovf.i8</code>
<code>nuint</code>	<code>ulong</code>	ImplicitNumeric	<code>conv.u8</code> / <code>conv.ovf.u8</code>
<code>nuint</code>	<code>char</code>	ExplicitNumeric	<code>conv.u2</code> / <code>conv.ovf.u2.un</code>
<code>nuint</code>	<code>float</code>	ImplicitNumeric	<code>conv.r.un</code> <code>conv.r4</code>
<code>nuint</code>	<code>double</code>	ImplicitNumeric	<code>conv.r.un</code> <code>conv.r8</code>
<code>nuint</code>	<code>decimal</code>	ImplicitNumeric	<code>conv.u8 decimal</code> <code>decimal.op_implicit(ulong)</code>
<code>nuint</code>	<code>IntPtr</code>	None	
<code>nuint</code>	<code>UIntPtr</code>	Identity	

OPERAND	TARGET	CONVERSION	IL
<code>nuint</code>	Enumeration	ExplicitEnumeration	

Conversion from `A` to `Nullable` is:

- an implicit nullable conversion if there is an identity conversion or implicit conversion from `A` to `B`;
- an explicit nullable conversion if there is an explicit conversion from `A` to `B`;
- otherwise invalid.

Conversion from `Nullable<A>` to `B` is:

- an explicit nullable conversion if there is an identity conversion or implicit or explicit numeric conversion from `A` to `B`;
- otherwise invalid.

Conversion from `Nullable<A>` to `Nullable` is:

- an identity conversion if there is an identity conversion from `A` to `B`;
- an explicit nullable conversion if there is an implicit or explicit numeric conversion from `A` to `B`;
- otherwise invalid.

Operators

The predefined operators are as follows. These operators are considered during overload resolution based on normal rules for implicit conversions *if at least one of the operands is of type `nint` or `nuint`*.

(The IL for each operator includes the variants for `unchecked` and `checked` contexts if different.)

UNARY	OPERATOR SIGNATURE	IL
<code>+</code>	<code>nint operator +(nint value)</code>	<code>nop</code>
<code>+</code>	<code>nuint operator +(nuint value)</code>	<code>nop</code>
<code>-</code>	<code>nint operator -(nint value)</code>	<code>neg</code>
<code>~</code>	<code>nint operator ~(nint value)</code>	<code>not</code>
<code>~</code>	<code>nuint operator ~(nuint value)</code>	<code>not</code>

BINARY	OPERATOR SIGNATURE	IL
<code>+</code>	<code>nint operator +(nint left, nint right)</code>	<code>add</code> / <code>add.ovf</code>
<code>+</code>	<code>nuint operator +(nuint left, nuint right)</code>	<code>add</code> / <code>add.ovf.un</code>
<code>-</code>	<code>nint operator -(nint left, nint right)</code>	<code>sub</code> / <code>sub.ovf</code>
<code>-</code>	<code>nuint operator -(nuint left, nuint right)</code>	<code>sub</code> / <code>sub.ovf.un</code>

BINARY	OPERATOR SIGNATURE	IL
<code>*</code>	<code>nint operator *(nint left, nint right)</code>	<code>mul</code> / <code>mul.ovf</code>
<code>*</code>	<code>nuint operator *(nuint left, nuint right)</code>	<code>mul</code> / <code>mul.ovf.un</code>
<code>/</code>	<code>nint operator /(nint left, nint right)</code>	<code>div</code>
<code>/</code>	<code>nuint operator /(nuint left, nuint right)</code>	<code>div.un</code>
<code>%</code>	<code>nint operator %(nint left, nint right)</code>	<code>rem</code>
<code>%</code>	<code>nuint operator %(nuint left, nuint right)</code>	<code>rem.un</code>
<code>==</code>	<code>bool operator ==(nint left, nint right)</code>	<code>beq</code> / <code>ceq</code>
<code>==</code>	<code>bool operator ==(nuint left, nuint right)</code>	<code>beq</code> / <code>ceq</code>
<code>!=</code>	<code>bool operator !=(nint left, nint right)</code>	<code>bne</code>
<code>!=</code>	<code>bool operator !=(nuint left, nuint right)</code>	<code>bne</code>
<code><</code>	<code>bool operator <(nint left, nint right)</code>	<code>blt</code> / <code>clt</code>
<code><</code>	<code>bool operator <(nuint left, nuint right)</code>	<code>blt.un</code> / <code>clt.un</code>
<code><=</code>	<code>bool operator <=(nint left, nint right)</code>	<code>ble</code>
<code><=</code>	<code>bool operator <=(nuint left, nuint right)</code>	<code>ble.un</code>
<code>></code>	<code>bool operator >(nint left, nint right)</code>	<code>bgt</code> / <code>cgt</code>
<code>></code>	<code>bool operator >(nuint left, nuint right)</code>	<code>bgt.un</code> / <code>cgt.un</code>
<code>>=</code>	<code>bool operator >=(nint left, nint right)</code>	<code>bge</code>
<code>>=</code>	<code>bool operator >=(nuint left, nuint right)</code>	<code>bge.un</code>
<code>&</code>	<code>nint operator &(nint left, nint right)</code>	<code>and</code>

BINARY	OPERATOR SIGNATURE	IL
<code>&</code>	<code>nuint operator &(nuint left, nuint right)</code>	<code>and</code>
<code> </code>	<code>nint operator (nint left, nint right)</code>	<code>or</code>
<code> </code>	<code>nuint operator (nuint left, nuint right)</code>	<code>or</code>
<code>^</code>	<code>nint operator ^(nint left, nint right)</code>	<code>xor</code>
<code>^</code>	<code>nuint operator ^(nuint left, nuint right)</code>	<code>xor</code>
<code><<</code>	<code>nint operator <<(nint left, int right)</code>	<code>shl</code>
<code><<</code>	<code>nuint operator <<(nuint left, int right)</code>	<code>shl</code>
<code>>></code>	<code>nint operator >>(nint left, int right)</code>	<code>shr</code>
<code>>></code>	<code>nuint operator >>(nuint left, int right)</code>	<code>shr.un</code>

For some binary operators, the IL operators support additional operand types (see [ECMA-335](#) III.1.5 Operand type table). But the set of operand types supported by C# is limited for simplicity and for consistency with existing operators in the language.

Lifted versions of the operators, where the arguments and return types are `nint?` and `nuint?`, are supported.

Compound assignment operations `x op= y` where `x` or `y` are native ints follow the same rules as with other primitive types with pre-defined operators. Specifically the expression is bound as `x = (T)(x op y)` where `T` is the type of `x` and where `x` is only evaluated once.

The shift operators should mask the number of bits to shift - to 5 bits if `sizeof(nint)` is 4, and to 6 bits if `sizeof(nint)` is 8. (see [shift operators](#) in C# spec).

The C#9 compiler will report errors binding to predefined native integer operators when compiling with an earlier language version, but will allow use of predefined conversions to and from native integers.

```
csc -langversion:9 -t:library A.cs
```

```
public class A
{
    public static nint F;
}
```

```
csc -langversion:8 -r:A.dll B.cs
```

```
class B : A
{
    static void Main()
    {
        F = F + 1; // error: nint operator+ not available with -langversion:8
        F = (System.IntPtr)F + 1; // ok
    }
}
```

Dynamic

The conversions and operators are synthesized by the compiler and are not part of the underlying `IntPtr` and `UIntPtr` types. As a result those conversions and operators *are not available* from the runtime binder for `dynamic`.

```
nint x = 2;
nint y = x + x; // ok
dynamic d = x;
nint z = d + x; // RuntimeBinderException: '+' cannot be applied 'System.IntPtr' and 'System.IntPtr'
```

Type members

The only constructor for `nint` or `uint` is the parameter-less constructor.

The following members of `System.IntPtr` and `System.UIntPtr` *are explicitly excluded* from `nint` or `uint`:

```
// constructors
// arithmetic operators
// implicit and explicit conversions
public static readonly IntPtr Zero; // use 0 instead
public static int Size { get; } // use sizeof() instead
public static IntPtr Add(IntPtr pointer, int offset);
public static IntPtr Subtract(IntPtr pointer, int offset);
public int ToInt32();
public long ToInt64();
public void* ToPointer();
```

The remaining members of `System.IntPtr` and `System.UIntPtr` *are implicitly included* in `nint` and `uint`. For .NET Framework 4.7.2:

```
public override bool Equals(object obj);
public override int GetHashCode();
public override string ToString();
public string ToString(string format);
```

Interfaces implemented by `System.IntPtr` and `System.UIntPtr` *are implicitly included* in `nint` and `uint`, with occurrences of the underlying types replaced by the corresponding native integer types. For instance if `IntPtr` implements `ISerializable`, `IEquatable<IntPtr>`, `IComparable<IntPtr>`, then `nint` implements `ISerializable`, `IEquatable<nint>`, `IComparable<nint>`.

Overriding, hiding, and implementing

`nint` and `System.IntPtr`, and `uint` and `System.UIntPtr`, are considered equivalent for overriding, hiding, and implementing.

Overloads cannot differ by `nint` and `System.IntPtr`, and `uint` and `System.UIntPtr`, alone. Overrides and implementations may differ by `nint` and `System.IntPtr`, or `uint` and `System.UIntPtr`, alone. Methods hide other methods that differ by `nint` and `System.IntPtr`, or `uint` and `System.UIntPtr`, alone.

Miscellaneous

`nint` and `nuint` expressions used as array indices are emitted without conversion.

```
static object GetItem(object[] array, nint index)
{
    return array[index]; // ok
}
```

`nint` and `nuint` can be used as an `enum` base type.

```
enum E : nint // ok
{
}
```

Reads and writes are atomic for types `nint`, `nuint`, and `enum` with base type `nint` or `nuint`.

Fields may be marked `volatile` for types `nint` and `nuint`. [ECMA-334](#) 15.5.4 does not include `enum` with base type `System.IntPtr` or `System.UIntPtr` however.

`default(nint)` and `new nint()` are equivalent to `(nint)0`.

`typeof(nint)` is `typeof(IntPtr)`.

`sizeof(nint)` is supported but requires compiling in an unsafe context (as does `sizeof(IntPtr)`). The value is not a compile-time constant. `sizeof(nint)` is implemented as `sizeof(IntPtr)` rather than `IntPtr.Size`.

Compiler diagnostics for type references involving `nint` or `nuint` report `nint` or `nuint` rather than `IntPtr` or `UIntPtr`.

Metadata

`nint` and `nuint` are represented in metadata as `System.IntPtr` and `System.UIntPtr`.

Type references that include `nint` or `nuint` are emitted with a `System.Runtime.CompilerServices.NativeIntegerAttribute` to indicate which parts of the type reference are native ints.

```

namespace System.Runtime.CompilerServices
{
    [AttributeUsage(
        AttributeTargets.Class |
        AttributeTargets.Event |
        AttributeTargets.Field |
        AttributeTargets.GenericParameter |
        AttributeTargets.Parameter |
        AttributeTargets.Property |
        AttributeTargets.ReturnValue,
        AllowMultiple = false,
        Inherited = false)]
    public sealed class NativeIntegerAttribute : Attribute
    {
        public NativeIntegerAttribute()
        {
            TransformFlags = new[] { true };
        }
        public NativeIntegerAttribute(bool[] flags)
        {
            TransformFlags = flags;
        }
        public readonly bool[] TransformFlags;
    }
}

```

The encoding of type references with `NativeIntegerAttribute` is covered in [NativeIntegerAttribute.md](#).

Alternatives

An alternative to the "type erasure" approach above is to introduce new types: `System.NativeInt` and `System.NativeUInt`.

```

public readonly struct NativeInt
{
    public IntPtr Value;
}

```

Distinct types would allow overloading distinct from `IntPtr` and would allow distinct parsing and `ToString()`. But there would be more work for the CLR to handle these types efficiently which defeats the primary purpose of the feature - efficiency. And interop with existing native int code that uses `IntPtr` would be more difficult.

Another alternative is to add more native int support for `IntPtr` in the framework but without any specific compiler support. Any new conversions and arithmetic operations would be supported by the compiler automatically. But the language would not provide keywords, constants, or `checked` operations.

Design meetings

- <https://github.com/dotnet/csharp-lang/blob/master/meetings/2017/LDM-2017-05-26.md>
- <https://github.com/dotnet/csharp-lang/blob/master/meetings/2017/LDM-2017-06-13.md>
- <https://github.com/dotnet/csharp-lang/blob/master/meetings/2017/LDM-2017-07-05.md#native-int-and-intptr-operators>
- <https://github.com/dotnet/csharp-lang/blob/master/meetings/2019/LDM-2019-10-23.md>
- <https://github.com/dotnet/csharp-lang/blob/master/meetings/2020/LDM-2020-03-25.md>

Function Pointers

12/28/2021 • 22 minutes to read • [Edit Online](#)

Summary

This proposal provides language constructs that expose IL opcodes that cannot currently be accessed efficiently, or at all, in C# today: `ldftn` and `calli`. These IL opcodes can be important in high performance code and developers need an efficient way to access them.

Motivation

The motivations and background for this feature are described in the following issue (as is a potential implementation of the feature):

<https://github.com/dotnet/csharplang/issues/191>

This is an alternate design proposal to [compiler intrinsics](#)

Detailed Design

Function pointers

The language will allow for the declaration of function pointers using the `delegate*` syntax. The full syntax is described in detail in the next section but it is meant to resemble the syntax used by `Func` and `Action` type declarations.

```
unsafe class Example {
    void Example(Action<int> a, delegate*<int, void> f) {
        a(42);
        f(42);
    }
}
```

These types are represented using the function pointer type as outlined in ECMA-335. This means invocation of a `delegate*` will use `calli` where invocation of a `delegate` will use `callvirt` on the `Invoke` method. Syntactically though invocation is identical for both constructs.

The ECMA-335 definition of method pointers includes the calling convention as part of the type signature (section 7.1). The default calling convention will be `managed`. Unmanaged calling conventions can be specified by putting an `unmanaged` keyword after the `delegate*` syntax, which will use the runtime platform default. Specific unmanaged conventions can then be specified in brackets to the `unmanaged` keyword by specifying any type starting with `CallConv` in the `System.Runtime.CompilerServices` namespace, leaving off the `CallConv` prefix. These types must come from the program's core library, and the set of valid combinations is platform-dependent.

```
//This method has a managed calling convention. This is the same as leaving the managed keyword off.
delegate* managed<int, int>;

// This method will be invoked using whatever the default unmanaged calling convention on the runtime
// platform is. This is platform and architecture dependent and is determined by the CLR at runtime.
delegate* unmanaged<int, int>;

// This method will be invoked using the cdecl calling convention
// Cdecl maps to System.Runtime.CompilerServices.CallConvCdecl
delegate* unmanaged[Cdecl] <int, int>;

// This method will be invoked using the stdcall calling convention, and suppresses GC transition
// Stdcall maps to System.Runtime.CompilerServices.CallConvStdcall
// SuppressGCTransition maps to System.Runtime.CompilerServices.CallConvSuppressGCTransition
delegate* unmanaged[Stdcall, SuppressGCTransition] <int, int>;
```

Conversions between `delegate*` types is done based on their signature including the calling convention.

```
unsafe class Example {
    void Conversions() {
        delegate*<int, int, int> p1 = ...;
        delegate* managed<int, int, int> p2 = ...;
        delegate* unmanaged<int, int, int> p3 = ...;

        p1 = p2; // okay p1 and p2 have compatible signatures
        Console.WriteLine(p2 == p1); // True
        p2 = p3; // error: calling conventions are incompatible
    }
}
```

A `delegate*` type is a pointer type which means it has all of the capabilities and restrictions of a standard pointer type:

- Only valid in an `unsafe` context.
- Methods which contain a `delegate*` parameter or return type can only be called from an `unsafe` context.
- Cannot be converted to `object`.
- Cannot be used as a generic argument.
- Can implicitly convert `delegate*` to `void*`.
- Can explicitly convert from `void*` to `delegate*`.

Restrictions:

- Custom attributes cannot be applied to a `delegate*` or any of its elements.
- A `delegate*` parameter cannot be marked as `params`.
- A `delegate*` type has all of the restrictions of a normal pointer type.
- Pointer arithmetic cannot be performed directly on function pointer types.

Function pointer syntax

The full function pointer syntax is represented by the following grammar:

```

pointer_type
: ...
| funcptr_type
;

funcptr_type
: 'delegate' '*' calling_convention_specifier? '<' funcptr_parameter_list funcptr_return_type '>'
;

calling_convention_specifier
: 'managed'
| 'unmanaged' ('[' unmanaged_calling_convention ']')?
;

unmanaged_calling_convention
: 'Cdecl'
| 'Stdcall'
| 'Thiscall'
| 'Fastcall'
| identifier (',' identifier)*
;

funcptr_parameter_list
: (funcptr_parameter ',' )*
;

funcptr_parameter
: funcptr_parameter_modifier? type
;

funcptr_return_type
: funcptr_return_modifier? return_type
;

funcptr_parameter_modifier
: 'ref'
| 'out'
| 'in'
;

funcptr_return_modifier
: 'ref'
| 'ref readonly'
;

```

If no `calling_convention_specifier` is provided, the default is `managed`. The precise metadata encoding of the `calling_convention_specifier` and what `identifier`s are valid in the `unmanaged_calling_convention` is covered in [Metadata Representation of Calling Conventions](#).

```

delegate int Func1(string s);
delegate Func1 Func2(Func1 f);

// Function pointer equivalent without calling convention
delegate* <string, int>;
delegate* <delegate* <string, int>, delegate* <string, int>>;

// Function pointer equivalent with calling convention
delegate* managed <string, int>;
delegate* <delegate* managed <string, int>, delegate* <string, int>>;

```

Function pointer conversions

In an unsafe context, the set of available implicit conversions (Implicit conversions) is extended to include the following implicit pointer conversions:

- *Existing conversions*

- From *funcptr_type* `F0` to another *funcptr_type* `F1`, provided all of the following are true:
 - `F0` and `F1` have the same number of parameters, and each parameter `D0n` in `F0` has the same `ref`, `out`, or `in` modifiers as the corresponding parameter `D1n` in `F1`.
 - For each value parameter (a parameter with no `ref`, `out`, or `in` modifier), an identity conversion, implicit reference conversion, or implicit pointer conversion exists from the parameter type in `F0` to the corresponding parameter type in `F1`.
 - For each `ref`, `out`, or `in` parameter, the parameter type in `F0` is the same as the corresponding parameter type in `F1`.
 - If the return type is by value (no `ref` or `ref readonly`), an identity, implicit reference, or implicit pointer conversion exists from the return type of `F1` to the return type of `F0`.
 - If the return type is by reference (`ref` or `ref readonly`), the return type and `ref` modifiers of `F1` are the same as the return type and `ref` modifiers of `F0`.
 - The calling convention of `F0` is the same as the calling convention of `F1`.

Allow address-of to target methods

Method groups will now be allowed as arguments to an address-of expression. The type of such an expression will be a `delegate*` which has the equivalent signature of the target method and a managed calling convention:

```
unsafe class Util {
    public static void Log() { }

    void Use() {
        delegate*<void> ptr1 = &Util.Log;

        // Error: type "delegate*<void>" not compatible with "delegate*<int>";
        delegate*<int> ptr2 = &Util.Log;
    }
}
```

In an unsafe context, a method `M` is compatible with a function pointer type `F` if all of the following are true:

- `M` and `F` have the same number of parameters, and each parameter in `M` has the same `ref`, `out`, or `in` modifiers as the corresponding parameter in `F`.
- For each value parameter (a parameter with no `ref`, `out`, or `in` modifier), an identity conversion, implicit reference conversion, or implicit pointer conversion exists from the parameter type in `M` to the corresponding parameter type in `F`.
- For each `ref`, `out`, or `in` parameter, the parameter type in `M` is the same as the corresponding parameter type in `F`.
- If the return type is by value (no `ref` or `ref readonly`), an identity, implicit reference, or implicit pointer conversion exists from the return type of `F` to the return type of `M`.
- If the return type is by reference (`ref` or `ref readonly`), the return type and `ref` modifiers of `F` are the same as the return type and `ref` modifiers of `M`.
- The calling convention of `M` is the same as the calling convention of `F`. This includes both the calling convention bit, as well as any calling convention flags specified in the unmanaged identifier.
- `M` is a static method.

In an unsafe context, an implicit conversion exists from an address-of expression whose target is a method group `E` to a compatible function pointer type `F` if `E` contains at least one method that is applicable in its normal form to an argument list constructed by use of the parameter types and modifiers of `F`, as described in the following.

- A single method `M` is selected corresponding to a method invocation of the form `E(A)` with the following modifications:
 - The arguments list `A` is a list of expressions, each classified as a variable and with the type and modifier (`ref`, `out`, or `in`) of the corresponding *funcptr_parameter_list* of `F`.
 - The candidate methods are only those methods that are applicable in their normal form, not those applicable in their expanded form.
 - The candidate methods are only those methods that are static.
- If the algorithm of overload resolution produces an error, then a compile-time error occurs. Otherwise, the algorithm produces a single best method `M` having the same number of parameters as `F` and the conversion is considered to exist.
- The selected method `M` must be compatible (as defined above) with the function pointer type `F`. Otherwise, a compile-time error occurs.
- The result of the conversion is a function pointer of type `F`.

This means developers can depend on overload resolution rules to work in conjunction with the address-of operator:

```
unsafe class Util {
    public static void Log() { }
    public static void Log(string p1) { }
    public static void Log(int i) { };

    void Use() {
        delegate*<void> a1 = &Log; // Log()
        delegate*<int, void> a2 = &Log; // Log(int i)

        // Error: ambiguous conversion from method group Log to "void*"
        void* v = &Log;
    }
}
```

The address-of operator will be implemented using the `ldftn` instruction.

Restrictions of this feature:

- Only applies to methods marked as `static`.
- Non-`static` local functions cannot be used in `&`. The implementation details of these methods are deliberately not specified by the language. This includes whether they are static vs. instance or exactly what signature they are emitted with.

Operators on Function Pointer Types

The section in unsafe code on operators is modified as such:

In an unsafe context, several constructs are available for operating on all `_pointer_type_s` that are not `_funcptr_type_s`:

- The `*` operator may be used to perform pointer indirection ([Pointer indirection](#)).
- The `->` operator may be used to access a member of a struct through a pointer ([Pointer member access](#)).
- The `[]` operator may be used to index a pointer ([Pointer element access](#)).
- The `&` operator may be used to obtain the address of a variable ([The address-of operator](#)).
- The `++` and `--` operators may be used to increment and decrement pointers ([Pointer increment and decrement](#)).
- The `+` and `-` operators may be used to perform pointer arithmetic ([Pointer arithmetic](#)).
- The `==`, `!=`, `<`, `>`, `<=`, and `=>` operators may be used to compare pointers ([Pointer comparison](#)).

- The `stackalloc` operator may be used to allocate memory from the call stack ([Fixed size buffers](#)).
- The `fixed` statement may be used to temporarily fix a variable so its address can be obtained ([The fixed statement](#)).

In an unsafe context, several constructs are available for operating on all `_funcptr_type_s`:

- The `&` operator may be used to obtain the address of static methods ([Allow address-of to target methods](#)).
- The `==`, `!=`, `<`, `>`, `<=`, and `=>` operators may be used to compare pointers ([Pointer comparison](#)).

Additionally, we modify all the sections in `Pointers in expressions` to forbid function pointer types, except `Pointer comparison` and `The sizeof operator`.

Better function member

The better function member specification will be changed to include the following line:

A `delegate*` is more specific than `void*`

This means that it is possible to overload on `void*` and a `delegate*` and still sensibly use the address-of operator.

Type Inference

In unsafe code, the following changes are made to the type inference algorithms:

Input types

<https://github.com/dotnet/csharpplang/blob/master/spec/expressions.md#input-types>

The following is added:

If `E` is an address-of method group and `T` is a function pointer type then all the parameter types of `T` are input types of `E` with type `T`.

Output types

<https://github.com/dotnet/csharpplang/blob/master/spec/expressions.md#output-types>

The following is added:

If `E` is an address-of method group and `T` is a function pointer type then the return type of `T` is an output type of `E` with type `T`.

Output type inferences

<https://github.com/dotnet/csharpplang/blob/master/spec/expressions.md#output-type-inferences>

The following bullet is added between bullets 2 and 3:

- If `E` is an address-of method group and `T` is a function pointer type with parameter types `T1...Tk` and return type `Tb`, and overload resolution of `E` with the types `T1...Tk` yields a single method with return type `U`, then a *lower-bound inference* is made from `U` to `Tb`.

Exact inferences

<https://github.com/dotnet/csharpplang/blob/master/spec/expressions.md#exact-inferences>

The following sub-bullet is added as a case to bullet 2:

- `V` is a function pointer type `delegate*<V2...Vk, V1>` and `U` is a function pointer type

`delegate*<U2..Uk, U1>` , and the calling convention of `v` is identical to `u` , and the refness of `vi` is identical to `ui` .

Lower-bound inferences

<https://github.com/dotnet/csharpplang/blob/master/spec/expressions.md#lower-bound-inferences>

The following case is added to bullet 3:

- `v` is a function pointer type `delegate*<V2..Vk, V1>` and there is a function pointer type `delegate*<U2..Uk, U1>` such that `u` is identical to `delegate*<U2..Uk, U1>` , and the calling convention of `v` is identical to `u` , and the refness of `vi` is identical to `ui` .

The first bullet of inference from `ui` to `vi` is modified to:

- If `u` is not a function pointer type and `ui` is not known to be a reference type, or if `u` is a function pointer type and `ui` is not known to be a function pointer type or a reference type, then an *exact inference* is made

Then, added after the 3rd bullet of inference from `ui` to `vi` :

- Otherwise, if `v` is `delegate*<V2..Vk, V1>` then inference depends on the i-th parameter of `delegate*<V2..Vk, V1>` :
 - If `V1`:
 - If the return is by value, then a *lower-bound inference* is made.
 - If the return is by reference, then an *exact inference* is made.
 - If `V2..Vk`:
 - If the parameter is by value, then an *upper-bound inference* is made.
 - If the parameter is by reference, then an *exact inference* is made.

Upper-bound inferences

<https://github.com/dotnet/csharpplang/blob/master/spec/expressions.md#upper-bound-inferences>

The following case is added to bullet 2:

- `u` is a function pointer type `delegate*<U2..Uk, U1>` and `v` is a function pointer type which is identical to `delegate*<V2..Vk, V1>` , and the calling convention of `u` is identical to `v` , and the refness of `ui` is identical to `vi` .

The first bullet of inference from `ui` to `vi` is modified to:

- If `u` is not a function pointer type and `ui` is not known to be a reference type, or if `u` is a function pointer type and `ui` is not known to be a function pointer type or a reference type, then an *exact inference* is made

Then added after the 3rd bullet of inference from `ui` to `vi` :

- Otherwise, if `u` is `delegate*<U2..Uk, U1>` then inference depends on the i-th parameter of `delegate*<U2..Uk, U1>` :
 - If `U1`:
 - If the return is by value, then an *upper-bound inference* is made.
 - If the return is by reference, then an *exact inference* is made.

- If U2..Uk:
 - If the parameter is by value, then a *lower-bound inference* is made.
 - If the parameter is by reference, then an *exact inference* is made.

Metadata representation of `in`, `out`, and `ref readonly` parameters and return types

Function pointer signatures have no parameter flags location, so we must encode whether parameters and the return type are `in`, `out`, or `ref readonly` by using modreqs.

`in`

We reuse `System.Runtime.InteropServices.InAttribute`, applied as a `modreq` to the ref specifier on a parameter or return type, to mean the following:

- If applied to a parameter ref specifier, this parameter is treated as `in`.
- If applied to the return type ref specifier, the return type is treated as `ref readonly`.

`out`

We use `System.Runtime.InteropServices.OutAttribute`, applied as a `modreq` to the ref specifier on a parameter type, to mean that the parameter is an `out` parameter.

Errors

- It is an error to apply `outAttribute` as a modreq to a return type.
- It is an error to apply both `InAttribute` and `OutAttribute` as a modreq to a parameter type.
- If either are specified via modopt, they are ignored.

Metadata Representation of Calling Conventions

Calling conventions are encoded in a method signature in metadata by a combination of the `CallKind` flag in the signature and zero or more `modopt`s at the start of the signature. ECMA-335 currently declares the following elements in the `CallKind` flag:

```
CallKind
: default
| unmanaged cdecl
| unmanaged fastcall
| unmanaged thiscall
| unmanaged stdcall
| varargs
;
```

Of these, function pointers in C# will support all but `varargs`.

In addition, the runtime (and eventually 335) will be updated to include a new `CallKind` on new platforms. This does not have a formal name currently, but this document will use `unmanaged ext` as a placeholder to stand for the new extensible calling convention format. With no `modopt`s, `unmanaged ext` is the platform default calling convention, `unmanaged` without the square brackets.

Mapping the `calling_convention_specifier` to a `CallKind`

A `calling_convention_specifier` that is omitted, or specified as `managed`, maps to the `default` `CallKind`. This is default `CallKind` of any method not attributed with `UnmanagedCallersOnly`.

C# recognizes 4 special identifiers that map to specific existing unmanaged `CallKind`s from ECMA 335. In order for this mapping to occur, these identifiers must be specified on their own, with no other identifiers, and this

requirement is encoded into the spec for `unmanaged_calling_convention`s. These identifiers are `Cdecl`, `Thiscall`, `Stdcall`, and `Fastcall`, which correspond to `unmanaged cdecl`, `unmanaged thiscall`, `unmanaged stdcall`, and `unmanaged fastcall`, respectively. If more than one `identifier` is specified, or the single `identifier` is not of the specially recognized identifiers, we perform special name lookup on the identifier with the following rules:

- We prepend the `identifier` with the string `CallConv`.
- We look only at types defined in the `System.Runtime.CompilerServices` namespace.
- We look only at types defined in the core library of the application, which is the library that defines `System.Object` and has no dependencies.
- We look only at public types.

If lookup succeeds on all of the `identifier`s specified in an `unmanaged_calling_convention`, we encode the `CallKind` as `unmanaged ext`, and encode each of the resolved types in the set of `modopt`s at the beginning of the function pointer signature. As a note, these rules mean that users cannot prefix these `identifier`s with `CallConv`, as that will result in looking up `CallConvCallConvVectorCall`.

When interpreting metadata, we first look at the `CallKind`. If it is anything other than `unmanaged ext`, we ignore all `modopt`s on the return type for the purposes of determining the calling convention, and use only the `CallKind`. If the `CallKind` is `unmanaged ext`, we look at the `modopts` at the start of the function pointer type, taking the union of all types that meet the following requirements:

- The is defined in the core library, which is the library that references no other libraries and defines `System.Object`.
- The type is defined in the `System.Runtime.CompilerServices` namespace.
- The type starts with the prefix `CallConv`.
- The type is public.

These represent the types that must be found when performing lookup on the `identifier`s in an `unmanaged_calling_convention` when defining a function pointer type in source.

It is an error to attempt to use a function pointer with a `CallKind` of `unmanaged ext` if the target runtime does not support the feature. This will be determined by looking for the presence of the `System.Runtime.CompilerServices.RuntimeFeature.UnmanagedCallKind` constant. If this constant is present, the runtime is considered to support the feature.

`System.Runtime.InteropServices.UnmanagedCallersOnlyAttribute`

`System.Runtime.InteropServices.UnmanagedCallersOnlyAttribute` is an attribute used by the CLR to indicate that a method should be called with a specific calling convention. Because of this, we introduce the following support for working with the attribute:

- It is an error to directly call a method annotated with this attribute from C#. Users must obtain a function pointer to the method and then invoke that pointer.
- It is an error to apply the attribute to anything other than an ordinary static method or ordinary static local function. The C# compiler will mark any non-static or static non-ordinary methods imported from metadata with this attribute as unsupported by the language.
- It is an error for a method marked with the attribute to have a parameter or return type that is not an `unmanaged_type`.
- It is an error for a method marked with the attribute to have type parameters, even if those type parameters are constrained to `unmanaged`.
- It is an error for a method in a generic type to be marked with the attribute.
- It is an error to convert a method marked with the attribute to a delegate type.
- It is an error to specify any types for `UnmanagedCallersOnly.CallConvs` that do not meet the requirements for

calling convention `modopts` in metadata.

When determining the calling convention of a method marked with a valid `UnmanagedCallersOnly` attribute, the compiler performs the following checks on the types specified in the `CallConvs` property to determine the effective `CallKind` and `modopts` that should be used to determine the calling convention:

- If no types are specified, the `CallKind` is treated as `unmanaged_ext`, with no calling convention `modopts` at the start of the function pointer type.
- If there is one type specified, and that type is named `CallConvCdecl`, `CallConvThiscall`, `CallConvStdcall`, or `CallConvFastcall`, the `CallKind` is treated as `unmanaged_cdecl`, `unmanaged_thiscall`, `unmanaged_stdcall`, or `unmanaged_fastcall`, respectively, with no calling convention `modopts` at the start of the function pointer type.
- If multiple types are specified or the single type is not named one of the specially called out types above, the `CallKind` is treated as `unmanaged_ext`, with the union of the types specified treated as `modopts` at the start of the function pointer type.

The compiler then looks at this effective `CallKind` and `modopt` collection and uses normal metadata rules to determine the final calling convention of the function pointer type.

Open Questions

Detecting runtime support for `unmanaged_ext`

<https://github.com/dotnet/runtime/issues/38135> tracks adding this flag. Depending on the feedback from review, we will either use the property specified in the issue, or use the presence of

`UnmanagedCallersOnlyAttribute` as the flag that determines whether the runtimes supports `unmanaged_ext`.

Considerations

Allow instance methods

The proposal could be extended to support instance methods by taking advantage of the `EXPLICITTHIS` CLI calling convention (named `instance` in C# code). This form of CLI function pointers puts the `this` parameter as an explicit first parameter of the function pointer syntax.

```
unsafe class Instance {
    void Use() {
        delegate* instance<Instance, string> f = &ToString;
        f(this);
    }
}
```

This is sound but adds some complication to the proposal. Particularly because function pointers which differed by the calling convention `instance` and `managed` would be incompatible even though both cases are used to invoke managed methods with the same C# signature. Also in every case considered where this would be valuable to have there was a simple work around: use a `static` local function.

```
unsafe class Instance {
    void Use() {
        static string toString(Instance i) => i.ToString();
        delegate*<Instance, string> f = &toString;
        f(this);
    }
}
```

Don't require unsafe at declaration

Instead of requiring `unsafe` at every use of a `delegate*`, only require it at the point where a method group is converted to a `delegate*`. This is where the core safety issues come into play (knowing that the containing assembly cannot be unloaded while the value is alive). Requiring `unsafe` on the other locations can be seen as excessive.

This is how the design was originally intended. But the resulting language rules felt very awkward. It's impossible to hide the fact that this is a pointer value and it kept peeking through even without the `unsafe` keyword. For example the conversion to `object` can't be allowed, it can't be a member of a `class`, etc ... The C# design is to require `unsafe` for all pointer uses and hence this design follows that.

Developers will still be capable of presenting a *safe* wrapper on top of `delegate*` values the same way that they do for normal pointer types today. Consider:

```
unsafe struct Action {
    delegate*<void> _ptr;

    Action(delegate*<void> ptr) => _ptr = ptr;
    public void Invoke() => _ptr();
}
```

Using delegates

Instead of using a new syntax element, `delegate*`, simply use existing `delegate` types with a `*` following the type:

```
Func<object, object, bool>* ptr = &object.ReferenceEquals;
```

Handling calling convention can be done by annotating the `delegate` types with an attribute that specifies a `CallingConvention` value. The lack of an attribute would signify the managed calling convention.

Encoding this in IL is problematic. The underlying value needs to be represented as a pointer yet it also must:

1. Have a unique type to allow for overloads with different function pointer types.
2. Be equivalent for OHI purposes across assembly boundaries.

The last point is particularly problematic. This means that every assembly which uses `Func<int>*` must encode an equivalent type in metadata even though `Func<int>*` is defined in an assembly though doesn't control. Additionally any other type which is defined with the name `System.Func<T>` in an assembly that is not `mscorlib` must be different than the version defined in `mscorlib`.

One option that was explored was emitting such a pointer as `mod_req(Func<int>) void*`. This doesn't work though as a `mod_req` cannot bind to a `TypeSpec` and hence cannot target generic instantiations.

Named function pointers

The function pointer syntax can be cumbersome, particularly in complex cases like nested function pointers. Rather than have developers type out the signature every time the language could allow for named declarations of function pointers as is done with `delegate`.

```
func* void Action();

unsafe class NamedExample {
    void M(Action a) {
        a();
    }
}
```

Part of the problem here is the underlying CLI primitive doesn't have names hence this would be purely a C# invention and require a bit of metadata work to enable. That is doable but is a significant amount of work. It essentially requires C# to have a companion to the type def table purely for these names.

Also when the arguments for named function pointers were examined we found they could apply equally well to a number of other scenarios. For example it would be just as convenient to declare named tuples to reduce the need to type out the full signature in all cases.

```
(int x, int y) Point;

class NamedTupleExample {
    void M(Point p) {
        Console.WriteLine(p.x);
    }
}
```

After discussion we decided to not allow named declaration of `delegate*` types. If we find there is significant need for this based on customer usage feedback then we will investigate a naming solution that works for function pointers, tuples, generics, etc ... This is likely to be similar in form to other suggestions like full `typedef` support in the language.

Future Considerations

static delegates

This refers to [the proposal](#) to allow for the declaration of `delegate` types which can only refer to `static` members. The advantage being that such `delegate` instances can be allocation free and better in performance sensitive scenarios.

If the function pointer feature is implemented the `static delegate` proposal will likely be closed out. The proposed advantage of that feature is the allocation free nature. However recent investigations have found that is not possible to achieve due to assembly unloading. There must be a strong handle from the `static delegate` to the method it refers to in order to keep the assembly from being unloaded out from under it.

To maintain every `static delegate` instance would be required to allocate a new handle which runs counter to the goals of the proposal. There were some designs where the allocation could be amortized to a single allocation per call-site but that was a bit complex and didn't seem worth the trade off.

That means developers essentially have to decide between the following trade offs:

1. Safety in the face of assembly unloading: this requires allocations and hence `delegate` is already a sufficient option.
2. No safety in face of assembly unloading: use a `delegate*`. This can be wrapped in a `struct` to allow usage outside an `unsafe` context in the rest of the code.

Suppress emitting of `localsinit` flag.

12/28/2021 • 3 minutes to read • [Edit Online](#)

Summary

Allow suppressing emit of `localsinit` flag via `SkipLocalsInitAttribute` attribute.

Motivation

Background

Per CLR spec local variables that do not contain references are not initialized to a particular value by the VM/JIT. Reading from such variables without initialization is type-safe, but otherwise the behavior is undefined and implementation specific. Typically uninitialized locals contain whatever values were left in the memory that is now occupied by the stack frame. That could lead to nondeterministic behavior and hard to reproduce bugs.

There are two ways to "assign" a local variable:

- by storing a value or
- by specifying `localsinit` flag which forces everything that is allocated from the local memory pool to be zero-initialized NOTE: this includes both local variables and `stackalloc` data.

Use of uninitialized data is discouraged and is not allowed in verifiable code. While it might be possible to prove that by the means of flow analysis, it is permitted for the verification algorithm to be conservative and simply require that `localsinit` is set.

Historically C# compiler emits `localsinit` flag on all methods that declare locals.

While C# employs definite-assignment analysis which is more strict than what CLR spec would require (C# also needs to consider scoping of locals), it is not strictly guaranteed that the resulting code would be formally verifiable:

- CLR and C# rules may not agree on whether passing a local as `out` argument is a `use`.
- CLR and C# rules may not agree on treatment of conditional branches when conditions are known (constant propagation).
- CLR could as well simply require `localinits`, since that is permitted.

Problem

In high-performance application the cost of forced zero-initialization could be noticeable. It is particularly noticeable when `stackalloc` is used.

In some cases JIT can elide initial zero-initialization of individual locals when such initialization is "killed" by subsequent assignments. Not all JITs do this and such optimization has limits. It does not help with `stackalloc`.

To illustrate that the problem is real - there is a known bug where a method not containing any `IL` locals would not have `localsinit` flag. The bug is already being exploited by users by putting `stackalloc` into such methods - intentionally to avoid initialization costs. That is despite the fact that absence of `IL` locals is an unstable metric and may vary depending on changes in codegen strategy. The bug should be fixed and users should get a more documented and reliable way of suppressing the flag.

Detailed design

Allow specifying `System.Runtime.CompilerServices.SkipLocalsInitAttribute` as a way to tell the compiler to not emit `localsinit` flag.

The end result of this will be that the locals may not be zero-initialized by the JIT, which is in most cases unobservable in C#.

In addition to that `stackalloc` data will not be zero-initialized. That is definitely observable, but also is the most motivating scenario.

Permitted and recognized attribute targets are: `Method`, `Property`, `Module`, `Class`, `Struct`, `Interface`, `Constructor`. However compiler will not require that attribute is defined with the listed targets nor it will care in which assembly the attribute is defined.

When attribute is specified on a container (`class`, `module`, containing method for a nested method, ...), the flag affects all methods contained within the container.

Synthesized methods "inherit" the flag from the logical container/owner.

The flag affects only codegen strategy for actual method bodies. I.E. the flag has no effect on abstract methods and is not propagated to overriding/implementing methods.

This is explicitly a *compiler feature* and *not a language feature*.

Similarly to compiler command line switches the feature controls implementation details of a particular codegen strategy and does not need to be required by the C# spec.

Drawbacks

- Old/other compilers may not honor the attribute. Ignoring the attribute is compatible behavior. Only may result in a slight perf hit.
- The code without `localinits` flag may trigger verification failures. Users that ask for this feature are generally unconcerned with verifiability.
- Applying the attribute at higher levels than an individual method has nonlocal effect, which is observable when `stackalloc` is used. Yet, this is the most requested scenario.

Alternatives

- omit `localinits` flag when method is declared in `unsafe` context. That could cause silent and dangerous behavior change from deterministic to nondeterministic in a case of `stackalloc`.
- omit `localinits` flag always. Even worse than above.
- omit `localinits` flag unless `stackalloc` is used in the method body. Does not address the most requested scenario and may turn code unverifiable with no option to revert that back.

Unresolved questions

- Should the attribute be actually emitted to metadata?

Design meetings

None yet.

Unconstrained type parameter annotations

12/28/2021 • 2 minutes to read • [Edit Online](#)

Summary

Allow nullable annotations for type parameters that are not constrained to value types or reference types: `T?`.

```
static T? FirstOrDefault<T>(this IEnumerable<T> collection) { ... }
```

`?` annotation

In C#8, `?` annotations could only be applied to type parameters that were explicitly constrained to value types or reference types. In C#9, `?` annotations can be applied to any type parameter, regardless of constraints.

Unless a type parameter is explicitly constrained to value types, annotations can only be applied within a `#nullable enable` context.

If a type parameter `T` is substituted with a reference type, then `T?` represents a nullable instance of that reference type.

```
var s1 = new string[0].FirstOrDefault(); // string? s1
var s2 = new string?[0].FirstOrDefault(); // string? s2
```

If `T` is substituted with a value type, then `T?` represents an instance of `T`.

```
var i1 = new int[0].FirstOrDefault(); // int i1
var i2 = new int?[0].FirstOrDefault(); // int? i2
```

If `T` is substituted with an annotated type `U?`, then `T?` represents the annotated type `U?` rather than `U??`.

```
var u1 = new U[0].FirstOrDefault(); // U? u1
var u2 = new U?[0].FirstOrDefault(); // U? u2
```

If `T` is substituted with a type `U`, then `T?` represents `U?`, even within a `#nullable disable` context.

```
#nullable disable
var u3 = new U[0].FirstOrDefault(); // U? u3
```

For return values, `T?` is equivalent to `[MaybeNull]T`; for argument values, `T?` is equivalent to `[AllowNull]T`. The equivalence is important when overriding or implementing interfaces from an assembly compiled with C#8.

```
public abstract class A
{
    [return: MaybeNull] public abstract T F1<T>();
    public abstract void F2<T>([AllowNull] T t);
}

public class B : A
{
    public override T? F1<T>() where T : default { ... } // matches A.F1<T>()
    public override void F2<T>(T? t) where T : default { ... } // matches A.F2<T>()
}
```

default constraint

For compatibility with existing code where overridden and explicitly implemented generic methods could not include explicit constraint clauses, `T?` in an overridden or explicitly implemented method is treated as `Nullable<T>` where `T` is a value type.

To allow annotations for type parameters constrained to reference types, C#8 allowed explicit `where T : class` and `where T : struct` constraints on the overridden or explicitly implemented method.

```
class A1
{
    public virtual void F1<T>(T? t) where T : struct { }
    public virtual void F1<T>(T? t) where T : class { }
}

class B1 : A1
{
    public override void F1<T>(T? t) /*where T : struct*/ { }
    public override void F1<T>(T? t) where T : class { }
}
```

To allow annotations for type parameters that are not constrained to reference types or value types, C#9 allows a new `where T : default` constraint.

```
class A2
{
    public virtual void F2<T>(T? t) where T : struct { }
    public virtual void F2<T>(T? t) { }
}

class B2 : A2
{
    public override void F2<T>(T? t) /*where T : struct*/ { }
    public override void F2<T>(T? t) where T : default { }
}
```

It is an error to use a `default` constraint other than on a method override or explicit implementation. It is an error to use a `default` constraint when the corresponding type parameter in the overridden or interface method is constrained to a reference type or value type.

Design meetings

- <https://github.com/dotnet/csharp-lang/blob/master/meetings/2019/LDM-2019-11-25.md>
- <https://github.com/dotnet/csharp-lang/blob/master/meetings/2020/LDM-2020-06-17.md#t>

Record structs

12/28/2021 • 10 minutes to read • [Edit Online](#)

The syntax for a record struct is as follows:

```
record_struct_declaration
    : attributes? struct_modifier* 'partial'? 'record' 'struct' identifier type_parameter_list?
      parameter_list? struct_interfaces? type_parameter_constraints_clause* record_struct_body
    ;

record_struct_body
    : struct_body
    | ';'
    ;
```

Record struct types are value types, like other struct types. They implicitly inherit from the class `System.ValueType`. The modifiers and members of a record struct are subject to the same restrictions as those of structs (accessibility on type, modifiers on members, `base(...)` instance constructor initializers, definite assignment for `this` in constructor, destructors, ...). Record structs will also follow the same rules as structs for parameterless instance constructors and field initializers, but this document assumes that we will lift those restrictions for structs generally.

See <https://github.com/dotnet/csharp-lang/blob/master/spec/structs.md> See [parameterless struct constructors spec](#).

Record structs cannot use `ref` modifier.

At most one partial type declaration of a partial record struct may provide a `parameter_list`. The `parameter_list` may not be empty.

Record struct parameters cannot use `ref`, `out` or `this` modifiers (but `in` and `params` are allowed).

Members of a record struct

In addition to the members declared in the record struct body, a record struct type has additional synthesized members. Members are synthesized unless a member with a "matching" signature is declared in the record struct body or an accessible concrete non-virtual member with a "matching" signature is inherited. Two members are considered matching if they have the same signature or would be considered "hiding" in an inheritance scenario. See <https://github.com/dotnet/csharp-lang/blob/master/spec/basic-concepts.md#signatures-and-overloading>

It is an error for a member of a record struct to be named "Clone".

It is an error for an instance field of a record struct to have an unsafe type.

A record struct is not permitted to declare a destructor.

The synthesized members are as follows:

Equality members

The synthesized equality members are similar as in a record class (`Equals` for this type, `Equals` for `object` type, `==` and `!=` operators for this type), except for the lack of `EqualityContract`, null checks or inheritance.

The record struct implements `System.IEquatable<R>` and includes a synthesized strongly-typed overload of `Equals(R other)` where `R` is the record struct. The method is `public`. The method can be declared explicitly. It is an error if the explicit declaration does not match the expected signature or accessibility.

If `Equals(R other)` is user-defined (not synthesized) but `GetHashCode` is not, a warning is produced.

```
public readonly bool Equals(R other);
```

The synthesized `Equals(R)` returns `true` if and only if for each instance field `fieldN` in the record struct the value of `System.Collections.Generic.EqualityComparer<TN>.Default.Equals(fieldN, other.fieldN)` where `TN` is the field type is `true`.

The record struct includes synthesized `==` and `!=` operators equivalent to operators declared as follows:

```
public static bool operator==(R r1, R r2)
    => r1.Equals(r2);
public static bool operator!=(R r1, R r2)
    => !(r1 == r2);
```

The `Equals` method called by the `==` operator is the `Equals(R other)` method specified above. The `!=` operator delegates to the `==` operator. It is an error if the operators are declared explicitly.

The record struct includes a synthesized override equivalent to a method declared as follows:

```
public override readonly bool Equals(object? obj);
```

It is an error if the override is declared explicitly. The synthesized override returns `other is R temp && Equals(temp)` where `R` is the record struct.

The record struct includes a synthesized override equivalent to a method declared as follows:

```
public override readonly int GetHashCode();
```

The method can be declared explicitly.

A warning is reported if one of `Equals(R)` and `GetHashCode()` is explicitly declared but the other method is not explicit.

The synthesized override of `GetHashCode()` returns an `int` result of combining the values of `System.Collections.Generic.EqualityComparer<TN>.Default.GetHashCode(fieldN)` for each instance field `fieldN` with `TN` being the type of `fieldN`.

For example, consider the following record struct:

```
record struct R1(T1 P1, T2 P2);
```

For this record struct, the synthesized equality members would be something like:

```

struct R1 : IEquatable<R1>
{
    public T1 P1 { get; set; }
    public T2 P2 { get; set; }
    public override bool Equals(object? obj) => obj is R1 temp && Equals(temp);
    public bool Equals(R1 other)
    {
        return
            EqualityComparer<T1>.Default.Equals(P1, other.P1) &&
            EqualityComparer<T2>.Default.Equals(P2, other.P2);
    }
    public static bool operator==(R1 r1, R1 r2)
        => r1.Equals(r2);
    public static bool operator!=(R1 r1, R1 r2)
        => !(r1 == r2);
    public override int GetHashCode()
    {
        return Combine(
            EqualityComparer<T1>.Default.GetHashCode(P1),
            EqualityComparer<T2>.Default.GetHashCode(P2));
    }
}

```

Printing members: PrintMembers and ToString methods

The record struct includes a synthesized method equivalent to a method declared as follows:

```
private bool PrintMembers(System.Text.StringBuilder builder);
```

The method does the following:

1. for each of the record struct's printable members (non-static public field and readable property members), appends that member's name followed by " = " followed by the member's value separated with ", ",
2. return true if the record struct has printable members.

For a member that has a value type, we will convert its value to a string representation using the most efficient method available to the target platform. At present that means calling `ToString` before passing to `StringBuilder.Append`.

If the record's printable members do not include a readable property with a non-`readonly` `get` accessor, then the synthesized `PrintMembers` is `readonly`. There is no requirement for the record's fields to be `readonly` for the `PrintMembers` method to be `readonly`.

The `PrintMembers` method can be declared explicitly. It is an error if the explicit declaration does not match the expected signature or accessibility.

The record struct includes a synthesized method equivalent to a method declared as follows:

```
public override string ToString();
```

If the record struct's `PrintMembers` method is `readonly`, then the synthesized `ToString()` method is `readonly`.

The method can be declared explicitly. It is an error if the explicit declaration does not match the expected signature or accessibility.

The synthesized method:

1. creates a `StringBuilder` instance,
2. appends the record struct name to the builder, followed by " { ",

- invokes the record struct's `PrintMembers` method giving it the builder, followed by `" "` if it returned true,
- appends `"}`",
- returns the builder's contents with `builder.ToString()`.

For example, consider the following record struct:

```
record struct R1(T1 P1, T2 P2);
```

For this record struct, the synthesized printing members would be something like:

```
struct R1 : IEquatable<R1>
{
    public T1 P1 { get; set; }
    public T2 P2 { get; set; }

    private bool PrintMembers(StringBuilder builder)
    {
        builder.Append(nameof(P1));
        builder.Append(" = ");
        builder.Append(this.P1); // or builder.Append(this.P1.ToString()); if P1 has a value type
        builder.Append(", ");

        builder.Append(nameof(P2));
        builder.Append(" = ");
        builder.Append(this.P2); // or builder.Append(this.P2.ToString()); if P2 has a value type

        return true;
    }

    public override string ToString()
    {
        var builder = new StringBuilder();
        builder.Append(nameof(R1));
        builder.Append(" { ");

        if (PrintMembers(builder))
            builder.Append(" ");

        builder.Append("}");
        return builder.ToString();
    }
}
```

Positional record struct members

In addition to the above members, record structs with a parameter list ("positional records") synthesize additional members with the same conditions as the members above.

Primary Constructor

A record struct has a public constructor whose signature corresponds to the value parameters of the type declaration. This is called the primary constructor for the type. It is an error to have a primary constructor and a constructor with the same signature already present in the struct.

Instance field declarations for a record struct are permitted to include variable initializers. If there is no primary constructor, the instance initializers execute as part of the parameterless constructor. Otherwise, at runtime the primary constructor executes the instance initializers appearing in the record-struct-body.

If a record struct has a primary constructor, any user-defined constructor must have an explicit `this` constructor initializer that calls the primary constructor or an explicitly declared constructor.

Parameters of the primary constructor as well as members of the record struct are in scope within initializers of instance fields or properties. Instance members would be an error in these locations (similar to how instance members are in scope in regular constructor initializers today, but an error to use), but the parameters of the primary constructor would be in scope and useable and would shadow members. Static members would also be useable.

A warning is produced if a parameter of the primary constructor is not read.

The definite assignment rules for struct instance constructors apply to the primary constructor of record structs. For instance, the following is an error:

```
record struct Pos(int X) // definite assignment error in primary constructor
{
    private int x;
    public int X { get { return x; } set { x = value; } } = X;
}
```

Properties

For each record struct parameter of a record struct declaration there is a corresponding public property member whose name and type are taken from the value parameter declaration.

For a record struct:

- A public `get` and `init` auto-property is created if the record struct has `readonly` modifier, `get` and `set` otherwise. Both kinds of set accessors (`set` and `init`) are considered "matching". So the user may declare an init-only property in place of a synthesized mutable one. An inherited `abstract` property with matching type is overridden. No auto-property is created if the record struct has an instance field with expected name and type. It is an error if the inherited property does not have `public` `get` and `set` / `init` accessors. It is an error if the inherited property or field is hidden.

The auto-property is initialized to the value of the corresponding primary constructor parameter. Attributes can be applied to the synthesized auto-property and its backing field by using `property:` or `field:` targets for attributes syntactically applied to the corresponding record struct parameter.

Deconstruct

A positional record struct with at least one parameter synthesizes a public void-returning instance method called `Deconstruct` with an out parameter declaration for each parameter of the primary constructor declaration. Each parameter of the `Deconstruct` method has the same type as the corresponding parameter of the primary constructor declaration. The body of the method assigns each parameter of the `Deconstruct` method to the value from an instance member access to a member of the same name. If the instance members accessed in the body do not include a property with a non-`readonly` `get` accessor, then the synthesized `Deconstruct` method is `readonly`. The method can be declared explicitly. It is an error if the explicit declaration does not match the expected signature or accessibility, or is static.

Allow `with` expression on structs

It is now valid for the receiver in a `with` expression to have a struct type.

On the right hand side of the `with` expression is a `member_initializer_list` with a sequence of assignments to *identifier*, which must be an accessible instance field or property of the receiver's type.

For a receiver with struct type, the receiver is first copied, then each `member_initializer` is processed the same way as an assignment to a field or property access of the result of the conversion. Assignments are processed in lexical order.

Improvements on records

Allow `record class`

The existing syntax for record types allows `record class` with the same meaning as `record`:

```
record_declaration
: attributes? class_modifier* 'partial'? 'record' 'class'? identifier type_parameter_list?
  parameter_list? record_base? type_parameter_constraints_clause* record_body
;
```

Allow user-defined positional members to be fields

See <https://github.com/dotnet/csharplang/blob/master/meetings/2020/LDM-2020-10-05.md#changing-the-member-type-of-a-primary-constructor-parameter>

No auto-property is created if the record has or inherits an instance field with expected name and type.

Allow parameterless constructors and member initializers in structs

See [parameterless struct constructors](#) spec.

Open questions

- how to recognize record structs in metadata? (we don't have an unspeakable clone method to leverage...)

Answered

- confirm that we want to keep PrintMembers design (separate method returning `bool`) (answer: yes)
- confirm we won't allow `record ref struct` (issue with `IEquatable<RefStruct>` and ref fields) (answer: yes)
- confirm implementation of equality members. Alternative is that synthesized `bool Equals(R other)`, `bool Equals(object? other)` and operators all just delegate to `ValueType.Equals`. (answer: yes)
- confirm that we want to allow field initializers when there is a primary constructor. Do we also want to allow parameterless struct constructors while we're at it (the Activator issue was apparently fixed)? (answer: yes, updated spec should be reviewed in LDM)
- how much do we want to say about `Combine` method? (answer: as little as possible)
- should we disallow a user-defined constructor with a copy constructor signature? (answer: no, there is no notion of copy constructor in the record structs spec)
- confirm that we want to disallow members named "Clone". (answer: correct)
- double-check that synthesized `Equals` logic is functionally equivalent to runtime implementation (e.g. `float.NaN`) (answer: confirmed in LDM)
- could field- or property-targeting attributes be placed in the positional parameter list? (answer: yes, same as for record class)
- `with` on generics? (answer: out of scope for C# 10)
- should `GetHashCode` include a hash of the type itself, to get different values between `record struct S1;` and `record struct S2;`? (answer: no)

Parameterless struct constructors

12/28/2021 • 6 minutes to read • [Edit Online](#)

Summary

Support parameterless constructors and instance field initializers for struct types.

Motivation

Explicit parameterless constructors would give more control over minimally constructed instances of the struct type. Instance field initializers would allow simplified initialization across multiple constructors. Together these would close an obvious gap between `struct` and `class` declarations.

Support for field initializers would also allow initialization of fields in `record struct` declarations without explicitly implementing the primary constructor.

```
record struct Person(string Name)
{
    public object Id { get; init; } = GetNextId();
}
```

If struct field initializers are supported for constructors with parameters, it seems natural to extend that to parameterless constructors as well.

```
record struct Person()
{
    public string Name { get; init; }
    public object Id { get; init; } = GetNextId();
}
```

Proposal

Instance field initializers

Instance field declarations for a struct may include initializers.

As with [class field initializers](#):

A variable initializer for an instance field cannot reference the instance being created.

Constructors

A struct may declare a parameterless instance constructor.

A parameterless instance constructor is valid for all struct kinds including `struct`, `readonly struct`, `ref struct`, and `record struct`.

If the struct declaration does not contain any explicit instance constructors, and the struct has field initializers, the compiler will synthesize a `public` parameterless instance constructor. The parameterless constructor may be synthesized even if all initializer values are zeros.

Otherwise, the struct (see [struct constructors](#)) ...

implicitly has a parameterless instance constructor which always returns the value that results from setting all value type fields to their default value and all reference type fields to null.

Modifiers

A parameterless instance struct constructor must be declared `public`.

```
struct S0 { } // ok
struct S1 { public S1() { } } // ok
struct S2 { internal S2() { } } // error: parameterless constructor must be 'public'
```

Non-public constructors are ignored when importing types from metadata.

Constructors can be declared `extern` or `unsafe`. Constructors cannot be `partial`.

Executing field initializers

Execution of struct instance field initializers matches execution of [class field initializers](#) with **one** qualifier:

When an instance constructor has no constructor initializer, **or when the constructor initializer `this()` represents the default parameterless constructor**, ... that constructor implicitly performs the initializations specified by the *variable_initializers* of the instance fields This corresponds to a sequence of assignments that are executed immediately upon entry to the constructor The variable initializers are executed in the textual order in which they appear in the ... declaration.

Definite assignment

Instance fields (other than `fixed` fields) must be definitely assigned in struct instance constructors that do not have a `this()` initializer (see [struct constructors](#)).

Definite assignment of struct instance fields is required within synthesized and explicit parameterless constructors.

```
struct S0 // ok: no synthesized constructor
{
    int x;
    object y;
}

struct S1
{
    int x = 1;
    object y; // error: field 'y' must be assigned
}

struct S2
{
    int x = 1;
    object y;
    public S2() { } // error: field 'y' must be assigned
}
```

No `base()` initializer

A `base()` initializer is disallowed in struct constructors.

The compiler will not emit a call to the base `System.ValueType` constructor from any struct instance constructors including explicit and synthesized parameterless constructors.

record struct

If a `record struct` does not contain a primary constructor nor any instance constructors, and the `record struct` has field initializers, the compiler will synthesize a `public` parameterless instance constructor.

```
record struct R0; // no parameterless .ctor
record struct R1 { int F = 42; } // synthesized .ctor: public R1() { F = 42; }
record struct R2(int F) { int F = F; } // no parameterless .ctor
```

A `record struct` with an empty parameter list will have a parameterless primary constructor.

```
record struct R3(); // primary .ctor: public R3() { }
record struct R4() { int F = 42; } // primary .ctor: public R4() { F = 42; }
```

An explicit parameterless constructor in a `record struct` must have a `this` initializer that calls the primary constructor or an explicitly declared constructor.

```
record struct R5(int F)
{
    public R5() { } // error: must have 'this' initializer that calls explicit .ctor
    public R5(object o) : this() { } // ok
    public int F = F;
}
```

Fields

The implicitly-defined parameterless constructor will zero fields rather than calling any parameterless constructors for the field types. No warnings are reported that field constructors are ignored. *No change from C#9.*

```
struct S0
{
    public S0() { }
}

struct S1
{
    S0 F; // S0 constructor ignored
}

struct S<T> where T : struct
{
    T F; // constructor (if any) ignored
}
```

`default` **expression**

`default` ignores the parameterless constructor and generates a zeroed instance. *No change from C#9.*

```
// struct S { public S() { } }

_ = default(S); // constructor ignored, no warning
```

`new()`

Object creation invokes the parameterless constructor if public; otherwise the instance is zeroed. *No change from C#9.*

```
// public struct PublicConstructor { public PublicConstructor() { } }
// public struct PrivateConstructor { private PrivateConstructor() { } }

_ = new PublicConstructor(); // call PublicConstructor::.ctor()
_ = new PrivateConstructor(); // initobj PrivateConstructor
```

A warning wave may report a warning for use of `new()` with a struct type that has constructors but no parameterless constructor. No warning will be reported when using substituting such a struct type for a type parameter with a `new()` or `struct` constraint.

```
struct S { public S(int i) { } }
static T CreateNew<T>() where T : new() => new T();

_ = new S(); // warning: no constructor called
_ = CreateNew<S>(); // ok
```

Uninitialized values

A local or field of a struct type that is not explicitly initialized is zeroed. The compiler reports a definite assignment error for an uninitialized struct that is not empty. *No change from C#9.*

```
NoConstructor s1;
PublicConstructor s2;
s1.ToString(); // error: use of unassigned local (unless type is empty)
s2.ToString(); // error: use of unassigned local (unless type is empty)
```

Array allocation

Array allocation ignores any parameterless constructor and generates zeroed elements. *No change from C#9.*

```
// struct S { public S() { } }

var a = new S[1]; // constructor ignored, no warning
```

Parameter default value `new()`

A parameter default value of `new()` binds to the parameterless constructor if public (and reports an error that the value is not constant); otherwise the instance is zeroed. *No change from C#9.*

```
// public struct PublicConstructor { public PublicConstructor() { } }
// public struct PrivateConstructor { private PrivateConstructor() { } }

static void F1(PublicConstructor s1 = new()) { } // error: default value must be constant
static void F2(PrivateConstructor s2 = new()) { } // ok: initobj
```

Type parameter constraints: `new()` and `struct`

The `new()` and `struct` type parameter constraints require the parameterless constructor to be `public` if defined (see [satisfying constraints](#)).

The compiler assumes all structs satisfy `new()` and `struct` constraints. *No change from C#9.*

```
// public struct PublicConstructor { public PublicConstructor() { } }
// public struct InternalConstructor { internal InternalConstructor() { } }

static T CreateNew<T>() where T : new() => new T();
static T CreateStruct<T>() where T : struct => new T();

_ = CreateNew<PublicConstructor>(); // ok
_ = CreateStruct<PublicConstructor>(); // ok

_ = CreateNew<InternalConstructor>(); // compiles; may fail at runtime
_ = CreateStruct<InternalConstructor>(); // compiles; may fail at runtime
```

`new T()` is emitted as a call to `System.Activator.CreateInstance<T>()`, and the compiler assumes the implementation of `CreateInstance<T>()` invokes the `public` parameterless constructor if defined.

With .NET Framework, `Activator.CreateInstance<T>()` invokes the parameterless constructor if the constraint is `where T : new()` but appears to ignore the parameterless constructor if the constraint is `where T : struct`.

Optional parameters

Constructors with optional parameters are not considered parameterless constructors. *No change from C#9.*

```
struct S1 { public S1(string s = "") { } }
struct S2 { public S2(params object[] args) { } }

_ = new S1(); // ok: ignores constructor
_ = new S2(); // ok: ignores constructor
```

Metadata

Explicit and synthesized parameterless struct instance constructors will be emitted to metadata.

Public parameterless struct instance constructors will be imported from metadata; non-public struct instance constructors will be ignored. *No change from C#9.*

See also

- <https://github.com/dotnet/roslyn/issues/1029>

Design meetings

- <https://github.com/dotnet/csharplang/blob/main/meetings/2021/LDM-2021-04-28.md#open-questions-in-record-and-parameterless-structs>
- <https://github.com/dotnet/csharplang/blob/main/meetings/2021/LDM-2021-03-10.md#parameterless-struct-constructors>
- <https://github.com/dotnet/csharplang/blob/main/meetings/2021/LDM-2021-01-27.md#field-initializers>

Global Using Directive

12/28/2021 • 13 minutes to read • [Edit Online](#)

Syntax for a using directive is extended with an optional `global` keyword that can precede the `using` keyword:

```
compilation_unit
    : extern_alias_directive* global_using_directive* using_directive* global_attributes?
  namespace_member_declaration*
    ;

global_using_directive
    : global_using_alias_directive
    | global_using_namespace_directive
    | global_using_static_directive
    ;

global_using_alias_directive
    : 'global' 'using' identifier '=' namespace_or_type_name ';'
    ;

global_using_namespace_directive
    : 'global' 'using' namespace_name ';'
    ;

global_using_static_directive
    : 'global' 'using' 'static' type_name ';'
    ;
```

- The *global_using_directives* are allowed only on the Compilation Unit level (cannot be used inside a *namespace_declaration*).
- The *global_using_directives*, if any, must precede any *using_directives*.
- The scope of a *global_using_directives* extends over the *namespace_member_declarations* of all compilation units within the program. The scope of a *global_using_directive* specifically does not include other *global_using_directives*. Thus, peer *global_using_directives* or those from a different compilation unit do not affect each other, and the order in which they are written is insignificant. The scope of a *global_using_directive* specifically does not include *using_directives* immediately contained in any compilation unit of the program.

The effect of adding a *global_using_directive* to a program can be thought of as the effect of adding a similar *using_directive* that resolves to the same target namespace or type to every compilation unit of the program. However, the target of a *global_using_directive* is resolved in context of the compilation unit that contains it.

Scopes

<https://github.com/dotnet/csharp-lang/blob/master/spec/basic-concepts.md#scopes>

These are the relevant bullet points with proposed additions (which are in **bold**):

- The scope of name defined by an *extern_alias_directive* extends over the ***global_using_directives***, *using_directives*, *global_attributes* and *namespace_member_declarations* of its immediately containing compilation unit or namespace body. An *extern_alias_directive* does not contribute any new members to the underlying declaration space. In other words, an *extern_alias_directive* is not transitive, but, rather, affects only the compilation unit or namespace body in which it occurs.
- The scope of a name defined or imported by a ***global_using_directive*** extends over the

global_attributes and *namespace_member_declarations* of all the *compilation_units* in the program.

Namespace and type names

<https://github.com/dotnet/csharpplang/blob/master/spec/basic-concepts.md#namespace-and-type-names>

Changes are made to the algorithm determining the meaning of a *namespace_or_type_name* as follows.

This is the relevant bullet point with proposed additions (which are in **bold**):

- If the *namespace_or_type_name* is of the form **I** or of the form **I<A1, ..., Ak>**:
 - If **K** is zero and the *namespace_or_type_name* appears within a generic method declaration (**Methods**) and if that declaration includes a type parameter (**Type parameters**) with name **I**, then the *namespace_or_type_name* refers to that type parameter.
 - Otherwise, if the *namespace_or_type_name* appears within a type declaration, then for each instance type **T** (**The instance type**), starting with the instance type of that type declaration and continuing with the instance type of each enclosing class or struct declaration (if any):
 - If **K** is zero and the declaration of **T** includes a type parameter with name **I**, then the *namespace_or_type_name* refers to that type parameter.
 - Otherwise, if the *namespace_or_type_name* appears within the body of the type declaration, and **T** or any of its base types contain a nested accessible type having name **I** and **K** type parameters, then the *namespace_or_type_name* refers to that type constructed with the given type arguments. If there is more than one such type, the type declared within the more derived type is selected. Note that non-type members (constants, fields, methods, properties, indexers, operators, instance constructors, destructors, and static constructors) and type members with a different number of type parameters are ignored when determining the meaning of the *namespace_or_type_name*.
 - If the previous steps were unsuccessful then, for each namespace **N**, starting with the namespace in which the *namespace_or_type_name* occurs, continuing with each enclosing namespace (if any), and ending with the global namespace, the following steps are evaluated until an entity is located:
 - If **K** is zero and **I** is the name of a namespace in **N**, then:
 - If the location where the *namespace_or_type_name* occurs is enclosed by a namespace declaration for **N** and the namespace declaration contains an *extern_alias_directive* or *using_alias_directive* that associates the name **I** with a namespace or type, **or any namespace declaration for N in the program contains a *global_using_alias_directive* that associates the name I with a namespace or type**, then the *namespace_or_type_name* is ambiguous and a compile-time error occurs.
 - Otherwise, the *namespace_or_type_name* refers to the namespace named **I** in **N**.
 - Otherwise, if **N** contains an accessible type having name **I** and **K** type parameters, then:
 - If **K** is zero and the location where the *namespace_or_type_name* occurs is enclosed by a namespace declaration for **N** and the namespace declaration contains an *extern_alias_directive* or *using_alias_directive* that associates the name **I** with a namespace or type, **or any namespace declaration for N in the program contains a *global_using_alias_directive* that associates the name I with a namespace or type**, then the *namespace_or_type_name* is ambiguous and a compile-time error occurs.
 - Otherwise, the *namespace_or_type_name* refers to the type constructed with the given type arguments.
 - Otherwise, if the location where the *namespace_or_type_name* occurs is enclosed by a namespace declaration for **N**:

- If `K` is zero and the namespace declaration contains an *extern_alias_directive* or *using_alias_directive* that associates the name `I` with an imported namespace or type, **or any namespace declaration for `N` in the program contains a *global_using_alias_directive* that associates the name `I` with an imported namespace or type**, then the *namespace_or_type_name* refers to that namespace or type.
- Otherwise, if the namespaces and type declarations imported by the *using_namespace_directives* and *using_alias_directives* of the namespace declaration **and the namespaces and type declarations imported by the *global_using_namespace_directives* and *global_using_static_directives* of any namespace declaration for `N` in the program** contain exactly one accessible type having name `I` and `K` type parameters, then the *namespace_or_type_name* refers to that type constructed with the given type arguments.
- Otherwise, if the namespaces and type declarations imported by the *using_namespace_directives* and *using_alias_directives* of the namespace declaration **and the namespaces and type declarations imported by the *global_using_namespace_directives* and *global_using_static_directives* of any namespace declaration for `N` in the program** contain more than one accessible type having name `I` and `K` type parameters, then the *namespace_or_type_name* is ambiguous and an error occurs.
- Otherwise, the *namespace_or_type_name* is undefined and a compile-time error occurs.

Simple names

<https://github.com/dotnet/csharp-lang/blob/master/spec/expressions.md#simple-names>

Changes are made to the *simple_name* evaluation rules as follows.

This is the relevant bullet point with proposed additions (which are **in bold**):

- Otherwise, for each namespace `N`, starting with the namespace in which the *simple_name* occurs, continuing with each enclosing namespace (if any), and ending with the global namespace, the following steps are evaluated until an entity is located:
 - If `K` is zero and `I` is the name of a namespace in `N`, then:
 - If the location where the *simple_name* occurs is enclosed by a namespace declaration for `N` and the namespace declaration contains an *extern_alias_directive* or *using_alias_directive* that associates the name `I` with a namespace or type, **or any namespace declaration for `N` in the program contains a *global_using_alias_directive* that associates the name `I` with a namespace or type**, then the *simple_name* is ambiguous and a compile-time error occurs.
 - Otherwise, the *simple_name* refers to the namespace named `I` in `N`.
 - Otherwise, if `N` contains an accessible type having name `I` and `K` type parameters, then:
 - If `K` is zero and the location where the *simple_name* occurs is enclosed by a namespace declaration for `N` and the namespace declaration contains an *extern_alias_directive* or *using_alias_directive* that associates the name `I` with a namespace or type, **or any namespace declaration for `N` in the program contains a *global_using_alias_directive* that associates the name `I` with a namespace or type**, then the *simple_name* is ambiguous and a compile-time error occurs.
 - Otherwise, the *namespace_or_type_name* refers to the type constructed with the given type arguments.
 - Otherwise, if the location where the *simple_name* occurs is enclosed by a namespace declaration for

N :

- If **K** is zero and the namespace declaration contains an *extern_alias_directive* or *using_alias_directive* that associates the name **I** with an imported namespace or type, or **any namespace declaration for N in the program contains a global_using_alias_directive** that associates the name **I** with an imported namespace or type, then the *simple_name* refers to that namespace or type.
- Otherwise, if the namespaces and type declarations imported by the *using_namespace_directives* and *using_static_directives* of the namespace declaration **and the namespaces and type declarations imported by the global_using_namespace_directives and global_using_static_directives of any namespace declaration for N in the program** contain exactly one accessible type or non-extension static member having name **I** and **K** type parameters, then the *simple_name* refers to that type or member constructed with the given type arguments.
- Otherwise, if the namespaces and types imported by the *using_namespace_directives* of the namespace declaration **and the namespaces and type declarations imported by the global_using_namespace_directives and global_using_static_directives of any namespace declaration for N in the program** contain more than one accessible type or non-extension-method static member having name **I** and **K** type parameters, then the *simple_name* is ambiguous and an error occurs.

Extension method invocations

<https://github.com/dotnet/csharpplang/blob/master/spec/expressions.md#extension-method-invocations>

Changes are made to the algorithm to find the best *type_name* **C** as follows. This is the relevant bullet point with proposed additions (which are in **bold**):

- Starting with the closest enclosing namespace declaration, continuing with each enclosing namespace declaration, and ending with the containing compilation unit, successive attempts are made to find a candidate set of extension methods:
 - If the given namespace or compilation unit directly contains non-generic type declarations **C_i** with eligible extension methods **M_j**, then the set of those extension methods is the candidate set.
 - If types **C_i** imported by *using_static_declarations* and directly declared in namespaces imported by *using_namespace_directives* in the given namespace or compilation unit **and, if containing compilation unit is reached, imported by global_using_static_declarations and directly declared in namespaces imported by global_using_namespace_directives in the program** directly contain eligible extension methods **M_j**, then the set of those extension methods is the candidate set.

Compilation units

<https://github.com/dotnet/csharpplang/blob/master/spec/namespaces.md#compilation-units>

A *compilation_unit* defines the overall structure of a source file. A compilation unit consists of **zero or more global_using_directives followed by zero or more using_directives followed by zero or more global_attributes followed by zero or more namespace_member_declarations**.

```
compilation_unit
: extern_alias_directive* global_using_directive* using_directive* global_attributes?
namespace_member_declaration*
;
```

A C# program consists of one or more compilation units, each contained in a separate source file. When a C#

program is compiled, all of the compilation units are processed together. Thus, compilation units can depend on each other, possibly in a circular fashion.

The *global_using_directives* of a compilation unit affect the *global_attributes* and *namespace_member_declarations* of all compilation units in the program.

Extern aliases

<https://github.com/dotnet/csharplang/blob/master/spec/namespaces.md#extern-aliases>

The scope of an *extern_alias_directive* extends over the *global_using_directives*, *using_directives*, *global_attributes* and *namespace_member_declarations* of its immediately containing compilation unit or namespace body.

Using alias directives

<https://github.com/dotnet/csharplang/blob/main/spec/namespaces.md#using-alias-directives>

The order in which *using_alias_directives* are written has no significance, and resolution of the *namespace_or_type_name* referenced by a *using_alias_directive* is not affected by the *using_alias_directive* itself or by other *using_directives* in the immediately containing compilation unit or namespace body, **and, if the *using_alias_directive* is immediately contained in a compilation unit, is not affected by the *global_using_directives* in the program.** In other words, the *namespace_or_type_name* of a *using_alias_directive* is resolved as if the immediately containing compilation unit or namespace body had no *using_directives* and, **if the *using_alias_directive* is immediately contained in a compilation unit, the program had no *global_using_directives*.** A *using_alias_directive* may however be affected by *extern_alias_directives* in the immediately containing compilation unit or namespace body.

Global Using alias directives

A *global_using_alias_directive* introduces an identifier that serves as an alias for a namespace or type within the program.

```
global_using_alias_directive
: 'global' 'using' identifier '=' namespace_or_type_name ';'
;
```

Within member declarations in any compilation unit of a program that contains a *global_using_alias_directive*, the identifier introduced by the *global_using_alias_directive* can be used to reference the given namespace or type.

The *identifier* of a *global_using_alias_directive* must be unique within the declaration space of any compilation unit of a program that contains the *global_using_alias_directive*.

Just like regular members, names introduced by *global_using_alias_directives* are hidden by similarly named members in nested scopes.

The order in which *global_using_alias_directives* are written has no significance, and resolution of the *namespace_or_type_name* referenced by a *global_using_alias_directive* is not affected by the *global_using_alias_directive* itself or by other *global_using_directives* or *using_directives* in the program. In other words, the *namespace_or_type_name* of a *global_using_alias_directive* is resolved as if the immediately containing compilation unit had no *using_directives* and the entire containing program had no *global_using_directives*. A *global_using_alias_directive* may however be affected by *extern_alias_directives* in the immediately containing compilation unit.

A *global_using_alias_directive* can create an alias for any namespace or type.

Accessing a namespace or type through an alias yields exactly the same result as accessing that namespace or type through its declared name.

Using aliases can name a closed constructed type, but cannot name an unbound generic type declaration without supplying type arguments.

Global Using namespace directives

A *global_using_namespace_directive* imports the types contained in a namespace into the program, enabling the identifier of each type to be used without qualification.

```
global_using_namespace_directive
: 'global' 'using' namespace_name ';'
;
```

Within member declarations in a program that contains a *global_using_namespace_directive*, the types contained in the given namespace can be referenced directly.

A *global_using_namespace_directive* imports the types contained in the given namespace, but specifically does not import nested namespaces.

Unlike a *global_using_alias_directive*, a *global_using_namespace_directive* may import types whose identifiers are already defined within a compilation unit of the program. In effect, in a given compilation unit, names imported by any *global_using_namespace_directive* in the program are hidden by similarly named members in the compilation unit.

When more than one namespace or type imported by *global_using_namespace_directives* or *global_using_static_directives* in the same program contain types by the same name, references to that name as a *type_name* are considered ambiguous.

Furthermore, when more than one namespace or type imported by *global_using_namespace_directives* or *global_using_static_directives* in the same program contain types or members by the same name, references to that name as a *simple_name* are considered ambiguous.

The *namespace_name* referenced by a *global_using_namespace_directive* is resolved in the same way as the *namespace_or_type_name* referenced by a *global_using_alias_directive*. Thus, *global_using_namespace_directives* in the same program do not affect each other and can be written in any order.

Global Using static directives

A *global_using_static_directive* imports the nested types and static members contained directly in a type declaration into the containing program, enabling the identifier of each member and type to be used without qualification.

```
global_using_static_directive
: 'global' 'using' 'static' type_name ';'
;
```

Within member declarations in a program that contains a *global_using_static_directive*, the accessible nested types and static members (except extension methods) contained directly in the declaration of the given type can be referenced directly.

A *global_using_static_directive* specifically does not import extension methods directly as static methods, but makes them available for extension method invocation.

A *global_using_static_directive* only imports members and types declared directly in the given type, not members and types declared in base classes.

Ambiguities between multiple *global_using_namespace_directives* and *global_using_static_directives* are discussed in the section for *global_using_namespace_directives* (above).

Namespace alias qualifiers

<https://github.com/dotnet/csharplang/blob/master/spec/namespaces.md#namespace-alias-qualifiers>

Changes are made to the algorithm determining the meaning of a *qualified_alias_member* as follows.

This is the relevant bullet point with proposed additions (which are in **bold**):

- Otherwise, starting with the namespace declaration ([Namespace declarations](#)) immediately containing the *qualified_alias_member* (if any), continuing with each enclosing namespace declaration (if any), and ending with the compilation unit containing the *qualified_alias_member*, the following steps are evaluated until an entity is located:
 - If the namespace declaration or compilation unit contains a *using_alias_directive* that associates N with a type, **or, when a compilation unit is reached, the program contains a *global_using_alias_directive* that associates N with a type**, then the *qualified_alias_member* is undefined and a compile-time error occurs.
 - Otherwise, if the namespace declaration or compilation unit contains an *extern_alias_directive* or *using_alias_directive* that associates N with a namespace, ***or, when a compilation unit is reached, the program contains a *global_using_alias_directive* that associates N with a namespace**, then:
 - If the namespace associated with N contains a namespace named I and K is zero, then the *qualified_alias_member* refers to that namespace.
 - Otherwise, if the namespace associated with N contains a non-generic type named I and K is zero, then the *qualified_alias_member* refers to that type.
 - Otherwise, if the namespace associated with N contains a type named I that has K type parameters, then the *qualified_alias_member* refers to that type constructed with the given type arguments.
 - Otherwise, the *qualified_alias_member* is undefined and a compile-time error occurs.

File Scoped Namespaces

12/28/2021 • 2 minutes to read • [Edit Online](#)

Summary

File scoped namespaces use a less verbose format for the typical case of files containing only one namespace. The file scoped namespace format is `namespace X.Y.Z;` (note the semicolon and lack of braces). This allows for files like the following:

```
namespace X.Y.Z;

using System;

class X
{
}
```

The semantics are that using the `namespace X.Y.Z;` form is equivalent to writing `namespace X.Y.Z { ... }` where the remainder of the file following the file-scoped namespace is in the `...` section of a standard namespace declaration.

Motivation

Analysis of the C# ecosystem shows that approximately 99.7% files are all of either one of these forms:

```
namespace X.Y.Z
{
    // usings

    // types
}
```

or

```
// usings

namespace X.Y.Z
{
    // types
}
```

However, both these forms force the user to indent the majority of their code and add a fair amount of ceremony for what is effectively a very basic concept. This affects clarity, uses horizontal and vertical space, and is often unsatisfying for users both used to C# and coming from other languages (which commonly have less ceremony here).

The primary goal of the feature therefore is to meet the needs of the majority of the ecosystem with less unnecessary boilerplate.

Detailed design

This proposal takes the form of a diff to the existing

<https://github.com/dotnet/csharplang/blob/main/spec/namespaces.md#compilation-units> section of the specification.

Diff

A *compilation_unit* defines the overall structure of a source file. A compilation unit consists of zero or more *using_directives* followed by zero or more *global_attributes* followed by zero or more *namespace_member_declarations*.

A *compilation_unit* defines the overall structure of a source file. A compilation unit consists of zero or more *using_directives* followed by zero or more *global_attributes* followed by a *compilation_unit_body*. A *compilation_unit_body* can either be a *file_scoped_namespace_declaration* or zero or more *statements* and *namespace_member_declarations*.

```
compilation_unit
~~      : extern_alias_directive* using_directive* global_attributes? namespace_member_declaration*~~
      : extern_alias_directive* using_directive* global_attributes? compilation_unit_body
      ;

compilation_unit_body
      : statement* namespace_member_declaration*
      | file_scoped_namespace_declaration
      ;
```

... unchanged ...

A *file_scoped_namespace_declaration* will contribute members corresponding to the *namespace_declaration* it is semantically equivalent to. See ([Namespace Declarations](#)) for more details.

Namespace declarations

A *namespace_declaration* consists of the keyword `namespace`, followed by a namespace name and body, optionally followed by a semicolon. A *file_scoped_namespace_declaration* consists of the keyword `namespace`, followed by a namespace name, a semicolon and an optional list of *extern_alias_directives*, *using_directives* and *type_declarations*.

```
namespace_declaration
      : 'namespace' qualified_identifier namespace_body ';' '?'
      ;

file_scoped_namespace_declaration
      : 'namespace' qualified_identifier ';' extern_alias_directive* using_directive* type_declaration*
      ;

... unchanged ...
```

... unchanged ...

the two namespace declarations above contribute to the same declaration space, in this case declaring two classes with the fully qualified names `N1.N2.A` and `N1.N2.B`. Because the two declarations contribute to the same declaration space, it would have been an error if each contained a declaration of a member with the same name.

A *file_scoped_namespace_declaration* permits a namespace declaration to be written without the `{ ... }` block. For example:


```
extern alias A;
namespace Name;
using B;
class C
{
}
```

is semantically equivalent to

```
extern alias A;
namespace Name
{
    using B;
    class C
    {
    }
}
```

Specifically, a *file_scoped_namespace_declaration* is treated the same as a *namespace_declaration* at the same location in the *compilation_unit* with the same *qualified_identifier*. The *extern_alias_directives*, *using_directives* and *type_declarations* of that *file_scoped_namespace_declaration* act as if they were declared in the same order inside the *namespace_body* of that *namespace_declaration*.

A source file cannot contain both a *file_scoped_namespace_declaration* and a *namespace_declaration*. A source file cannot contain multiple *file_scoped_namespace_declarations*. A *compilation_unit* cannot contain both a *file_scoped_namespace_declaration* and any top level *statements*. *type_declarations* cannot precede a *file_scoped_namespace_declaration*.

Extern aliases

... unchanged ...

Extended property patterns

12/28/2021 • 2 minutes to read • [Edit Online](#)

Summary

Allow property subpatterns to reference nested members, for instance:

```
if (e is MethodCallExpression { Method.Name: "MethodName" })
```

Instead of:

```
if (e is MethodCallExpression { Method: { Name: "MethodName" } })
```

Motivation

When you want to match a child property, nesting another recursive pattern adds too much noise which will hurt readability with no real advantage.

Detailed design

The *property_pattern* syntax is modified as follow:

```
property_pattern
: type? property_pattern_clause simple_designation?
;

property_pattern_clause
: '{' (subpattern (',' subpattern)* ','?)? '}'
;

subpattern
- : identifier ':' pattern
+ : subpattern_name ':' pattern
;

+subpattern_name
+ : identifier
+ | subpattern_name '.' identifier
+ ;
```

The receiver for each name lookup is the type of the previous member $T0$, starting from the *input type* of the *property_pattern*. if T is a nullable type, $T0$ is its underlying type, otherwise $T0$ is equal to T .

For example, a pattern of the form `{ Prop1.Prop2: pattern }` is exactly equivalent to

```
{ Prop1: { Prop2: pattern } }.
```

Note that this will include the null check when T is a nullable value type or a reference type. This null check means that the nested properties available will be the properties of $T0$, not of T .

Repeated member paths are allowed. The compilation of pattern matching can take advantage of common parts of patterns.

Improved Interpolated Strings

12/28/2021 • 28 minutes to read • [Edit Online](#)

Summary

We introduce a new pattern for creating and using interpolated string expressions to allow for efficient formatting and use in both general `string` scenarios and more specialized scenarios such as logging frameworks, without incurring unnecessary allocations from formatting the string in the framework.

Motivation

Today, string interpolation mainly lowers down to a call to `string.Format`. This, while general purpose, can be inefficient for a number of reasons:

1. It boxes any struct arguments, unless the runtime has happened to introduce an overload of `string.Format` that takes exactly the correct types of arguments in exactly the correct order.
 - This ordering is why the runtime is hesitant to introduce generic versions of the method, as it would lead to combinatoric explosion of generic instantiations of a very common method.
2. It has to allocate an array for the arguments in most cases.
3. There is no opportunity to avoid instantiating the instance if it's not needed. Logging frameworks, for example, will recommend avoiding string interpolation because it will cause a string to be realized that may not be needed, depending on the current log-level of the application.
4. It can never use `Span` or other ref struct types today, because ref structs are not allowed as generic type parameters, meaning that if a user wants to avoid copying to intermediate locations they have to manually format strings.

Internally, the runtime has a type called `ValueStringBuilder` to help deal with the first 2 of these scenarios. They pass a stackalloc'd buffer to the builder, repeatedly call `AppendFormat` with every part, and then get a final string out. If the resulting string goes past the bounds of the stack buffer, they can then move to an array on the heap. However, this type is dangerous to expose directly, as incorrect usage could lead to a rented array to be double-disposed, which then will cause all sorts of undefined behavior in the program as two locations think they have sole access to the rented array. This proposal creates a way to use this type safely from native C# code by just writing an interpolated string literal, leaving written code unchanged while improving every interpolated string that a user writes. It also extends this pattern to allow for interpolated strings passed as arguments to other methods to use a handler pattern, defined by receiver of the method, that will allow things like logging frameworks to avoid allocating strings that will never be needed, and giving C# users familiar, convenient interpolation syntax.

Detailed Design

The handler pattern

We introduce a new handler pattern that can represent an interpolated string passed as an argument to a method. The simple English of the pattern is as follows:

When an *interpolated_string_expression* is passed as an argument to a method, we look at the type of the parameter. If the parameter type has a constructor that can be invoked with 2 int parameters, `LiteralLength` and `FormattedCount`, optionally takes additional parameters specified by an attribute on the original parameter, optionally has an out boolean trailing parameter, and the type of the original parameter has instance `AppendLiteral` and `AppendFormatted` methods that can be invoked for every part of the interpolated string, then

we lower the interpolation using that, instead of into a traditional call to `string.Format(formatStr, args)`. A more concrete example is helpful for picturing this:

```
// The handler that will actually "build" the interpolated string"
[InterpolatedStringHandler]
public ref struct TraceLoggerParamsInterpolatedStringHandler
{
    // Storage for the built-up string

    private bool _logLevelEnabled;

    public TraceLoggerParamsInterpolatedStringHandler(int literalLength, int formattedCount, Logger logger,
out bool handlerIsValid)
    {
        if (!logger._logLevelEnabled)
        {
            handlerIsValid = false;
            return;
        }

        handlerIsValid = true;
        _logLevelEnabled = logger.EnabledLevel;
    }

    public void AppendLiteral(string s)
    {
        // Store and format part as required
    }

    public void AppendFormatted<T>(T t)
    {
        // Store and format part as required
    }
}

// The logger class. The user has an instance of this, accesses it via static state, or some other access
// mechanism
public class Logger
{
    // Initialization code omitted
    public LogLevel EnabledLevel;

    public void LogTrace([InterpolatedStringHandlerArguments("")]TraceLoggerParamsInterpolatedStringHandler
handler)
    {
        // Impl of logging
    }
}

Logger logger = GetLogger(LogLevel.Info);

// Given the above definitions, usage looks like this:
var name = "Fred Silberberg";
logger.LogTrace($"{name} will never be printed because info is < trace!");

// This is converted to:
var name = "Fred Silberberg";
var receiverTemp = logger;
var handler = new TraceLoggerParamsInterpolatedStringHandler(literalLength: 47, formattedCount: 1,
receiverTemp, out var handlerIsValid);
if (handlerIsValid)
{
    handler.AppendFormatted(name);
    handler.AppendLiteral(" will never be printed because info is < trace!");
}
receiverTemp.LogTrace(handler);
```

Here, because `TraceLoggerParamsInterpolatedStringHandler` has a constructor with the correct parameters, we say that the interpolated string has an implicit handler conversion to that parameter, and it lowers to the pattern shown above. The specese needed for this is a bit complicated, and is expanded below.

The rest of this proposal will use `Append...` to refer to either of `AppendLiteral` or `AppendFormatted` in cases when both are applicable.

New attributes

The compiler recognizes the `System.Runtime.CompilerServices.InterpolatedStringHandlerAttribute`:

```
using System;
namespace System.Runtime.CompilerServices
{
    [AttributeUsage(AttributeTargets.Class | AttributeTargets.Struct, AllowMultiple = false, Inherited = false)]
    public sealed class InterpolatedStringHandlerAttribute : Attribute
    {
        public InterpolatedStringHandlerAttribute()
        {
        }
    }
}
```

This attribute is used by the compiler to determine if a type is a valid interpolated string handler type.

The compiler also recognizes the `System.Runtime.CompilerServices.InterpolatedStringHandlerArgumentAttribute`:

```
namespace System.Runtime.CompilerServices
{
    [AttributeUsage(AttributeTargets.Parameter, AllowMultiple = false, Inherited = false)]
    public sealed class InterpolatedStringHandlerArgumentAttribute : Attribute
    {
        public InterpolatedHandlerArgumentAttribute(string argument);
        public InterpolatedHandlerArgumentAttribute(params string[] arguments);

        public string[] Arguments { get; }
    }
}
```

This attribute is used on parameters, to inform the compiler how to lower an interpolated string handler pattern used in a parameter position.

Interpolated string handler conversion

Type `T` is said to be an *applicable interpolated string handler type* if it is attributed with `System.Runtime.CompilerServices.InterpolatedStringHandlerAttribute`. There exists an implicit *interpolated string handler conversion* to `T` from an *interpolated string expression*, or an *additive expression* composed entirely of *interpolated string expression_s* and using only `+` operators.

For simplicity in the rest of this speclet, *interpolated string expression* refers to both a simple *interpolated string expression*, and to an *additive expression* composed entirely of *interpolated string expression_s* and using only `+` operators.

Note that this conversion always exists, regardless of whether there will be later errors when actually attempting to lower the interpolation using the handler pattern. This is done to help ensure that there are predictable and useful errors and that runtime behavior doesn't change based on the content of an interpolated string.

Applicable function member adjustments

We adjust the wording of the [applicable function member algorithm](#) as follows (a new sub-bullet is added to each section, in bold):

A function member is said to be an *applicable function member* with respect to an argument list `A` when all of the following are true:

- Each argument in `A` corresponds to a parameter in the function member declaration as described in [Corresponding parameters](#), and any parameter to which no argument corresponds is an optional parameter.
- For each argument in `A`, the parameter passing mode of the argument (i.e., value, `ref`, or `out`) is identical to the parameter passing mode of the corresponding parameter, and
 - for a value parameter or a parameter array, an implicit conversion ([Implicit conversions](#)) exists from the argument to the type of the corresponding parameter, or
 - for a `ref` parameter whose type is a struct type, an implicit *interpolated_string_handler_conversion* exists from the argument to the type of the corresponding parameter, or
 - for a `ref` or `out` parameter, the type of the argument is identical to the type of the corresponding parameter. After all, a `ref` or `out` parameter is an alias for the argument passed.

For a function member that includes a parameter array, if the function member is applicable by the above rules, it is said to be applicable in its *normal form*. If a function member that includes a parameter array is not applicable in its normal form, the function member may instead be applicable in its *expanded form*:

- The expanded form is constructed by replacing the parameter array in the function member declaration with zero or more value parameters of the element type of the parameter array such that the number of arguments in the argument list `A` matches the total number of parameters. If `A` has fewer arguments than the number of fixed parameters in the function member declaration, the expanded form of the function member cannot be constructed and is thus not applicable.
- Otherwise, the expanded form is applicable if for each argument in `A` the parameter passing mode of the argument is identical to the parameter passing mode of the corresponding parameter, and
 - for a fixed value parameter or a value parameter created by the expansion, an implicit conversion ([Implicit conversions](#)) exists from the type of the argument to the type of the corresponding parameter, or
 - for a `ref` parameter whose type is a struct type, an implicit *interpolated_string_handler_conversion* exists from the argument to the type of the corresponding parameter, or
 - for a `ref` or `out` parameter, the type of the argument is identical to the type of the corresponding parameter.

Important note: this means that if there are 2 otherwise equivalent overloads, that only differ by the type of the *applicable_interpolated_string_handler_type*, these overloads will be considered ambiguous. Further, because we do not see through explicit casts, it is possible that there could arise an unresolvable scenario where both applicable overloads use `InterpolatedStringHandlerArguments` and are totally uncalleable without manually performing the handler lowering pattern. We could potentially make changes to the better function member algorithm to resolve this if we so choose, but this scenario unlikely to occur and isn't a priority to address.

Better conversion from expression adjustments

We change the [better conversion from expression](#) section to the following:

Given an implicit conversion `c1` that converts from an expression `E` to a type `T1`, and an implicit conversion `c2` that converts from an expression `E` to a type `T2`, `c1` is a *better conversion* than `c2` if:

1. `E` is a non-constant *interpolated_string_expression*, `c1` is an *implicit_string_handler_conversion*, `T1` is an *applicable_interpolated_string_handler_type*, and `c2` is not an *implicit_string_handler_conversion*, or
2. `E` does not exactly match `T2` and at least one of the following holds:
 - `E` exactly matches `T1` ([Exactly matching Expression](#))
 - `T1` is a better conversion target than `T2` ([Better conversion target](#))

This does mean that there are some potentially non-obvious overload resolution rules, depending on whether the interpolated string in question is a constant-expression or not. For example:

```
void Log(string s) { ... }
void Log(TraceLoggerParamsInterpolatedStringHandler p) { ... }

Log(""); // Calls Log(string s), because "" is a constant expression
Log($"{test}"); // Calls Log(string s), because $"{test}" is a constant expression
Log($"{1}"); // Calls Log(TraceLoggerParamsInterpolatedStringHandler p), because $"{1}" is not a constant expression
```

This is introduced so that things that can simply be emitted as constants do so, and don't incur any overhead, while things that cannot be constant use the handler pattern.

InterpolatedStringHandler and Usage

We introduce a new type in `System.Runtime.CompilerServices`: `DefaultInterpolatedStringHandler`. This is a ref struct with many of the same semantics as `ValueStringBuilder`, intended for direct use by the C# compiler. This struct would look approximately like this:

```
// API Proposal issue: https://github.com/dotnet/runtime/issues/50601
namespace System.Runtime.CompilerServices
{
    [InterpolatedStringHandler]
    public ref struct DefaultInterpolatedStringHandler
    {
        public DefaultInterpolatedStringHandler(int literalLength, int formattedCount);
        public string ToStringAndClear();

        public void AppendLiteral(string value);

        public void AppendFormatted<T>(T value);
        public void AppendFormatted<T>(T value, string? format);
        public void AppendFormatted<T>(T value, int alignment);
        public void AppendFormatted<T>(T value, int alignment, string? format);

        public void AppendFormatted(ReadOnlySpan<char> value);
        public void AppendFormatted(ReadOnlySpan<char> value, int alignment = 0, string? format = null);

        public void AppendFormatted(string? value);
        public void AppendFormatted(string? value, int alignment = 0, string? format = null);

        public void AppendFormatted(object? value, int alignment = 0, string? format = null);
    }
}
```

We make a slight change to the rules for the meaning of an *interpolated_string_expression*:

If the type of an interpolated string is `string` and the type `System.Runtime.CompilerServices.DefaultInterpolatedStringHandler` exists, and the current context supports using that type, the string is lowered using the handler pattern. The final `string` value is then obtained by calling `ToStringAndClear()` on the handler type. Otherwise, if the type of an interpolated string is `System.IFormattable` or `System.FormattableString` [the rest is unchanged]

The "and the current context supports using that type" rule is intentionally vague to give the compiler leeway in optimizing usage of this pattern. The handler type is likely to be a ref struct type, and ref struct types are normally not permitted in async methods. For this particular case, the compiler would be allowed to make use the handler if none of the interpolation holes contain an `await` expression, as we can statically determine that the handler type is safely used without additional complicated analysis because the handler will be dropped after the interpolated string expression is evaluated.

Open Question:

Do we want to instead just make the compiler know about `DefaultInterpolatedStringHandler` and skip the `string.Format` call entirely? It would allow us to hide a method that we don't necessarily want to put in people's faces when they manually call `string.Format`.

Answer: Yes.

Open Question:

Do we want to have handlers for `System.IFormattable` and `System.FormattableString` as well?

Answer: No.

Handler pattern codegen

In this section, method invocation resolution refers to the steps listed [here](#).

Constructor resolution

Given an *applicable_interpolated_string_handler_type* `T` and an *interpolated_string_expression* `i`, method invocation resolution and validation for a valid constructor on `T` is performed as follows:

1. Member lookup for instance constructors is performed on `T`. The resulting method group is called `M`.
2. The argument list `A` is constructed as follows:
 - a. The first two arguments are integer constants, representing the literal length of `i`, and the number of *interpolation* components in `i`, respectively.
 - b. If `i` is used as an argument to some parameter `pi` in method `M1`, and parameter `pi` is attributed with `System.Runtime.CompilerServices.InterpolatedStringHandlerArgumentAttribute`, then for every name `Argx` in the `Arguments` array of that attribute the compiler matches it to a parameter `px` that has the same name. The empty string is matched to the receiver of `M1`.
 - If any `Argx` is not able to be matched to a parameter of `M1`, or an `Argx` requests the receiver of `M1` and `M1` is a static method, an error is produced and no further steps are taken.
 - Otherwise, the type of every resolved `px` is added to the argument list, in the order specified by the `Arguments` array. Each `px` is passed with the same `ref` semantics as is specified in `M1`.
 - c. The final argument is a `bool`, passed as an `out` parameter.
3. Traditional method invocation resolution is performed with method group `M` and argument list `A`. For the purposes of method invocation final validation, the context of `M` is treated as a *member_access* through type `T`.
 - If a single-best constructor `F` was found, the result of overload resolution is `F`.
 - If no applicable constructors were found, step 3 is retried, removing the final `bool` parameter from `A`. If this retry also finds no applicable members, an error is produced and no further steps are taken.
 - If no single-best method was found, the result of overload resolution is ambiguous, an error is produced, and no further steps are taken.
4. Final validation on `F` is performed.
 - If any element of `A` occurred lexically after `i`, an error is produced and no further steps are taken.
 - If any `A` requests the receiver of `F`, and `F` is an indexer being used as an *initializer_target* in a *member_initializer*, then an error is reported and no further steps are taken.

Note: the resolution here intentionally do *not* use the actual expressions passed as other arguments for `Argx` elements. We only consider the types post-conversion. This makes sure that we don't have double-conversion issues, or unexpected cases where a lambda is bound to one delegate type when passed to `M1` and bound to a different delegate type when passed to `M`.

Note: We report an error for indexers uses as member initializers because of the order of evaluation for nested

member initializers. Consider this code snippet:

```
var x1 = new C1 { C2 = { [GetString()] = { A = 2, B = 4 } } };

/* Lowering:
__c1 = new C1();
string argTemp = GetString();
__c1.C2[argTemp][1] = 2;
__c1.C2[argTemp][3] = 4;

Prints:
GetString
get_C2
get_C2
*/

string GetString()
{
    Console.WriteLine("GetString");
    return "";
}

class C1
{
    private C2 c2 = new C2();
    public C2 C2 { get { Console.WriteLine("get_C2"); return c2; } set { } }
}

class C2
{
    public C3 this[string s]
    {
        get => new C3();
        set { }
    }
}

class C3
{
    public int A
    {
        get => 0;
        set { }
    }
    public int B
    {
        get => 0;
        set { }
    }
}
```

The arguments to `__c1.c2[]` are evaluated *before* the receiver of the indexer. While we could come up with a lowering that works for this scenario (either by creating a temp for `__c1.c2` and sharing it across both indexer invocations, or only using it for the first indexer invocation and sharing the argument across both invocations) we think that any lowering would be confusing for what we believe is a pathological scenario. Therefore, we forbid the scenario entirely.

Open-Question:

If we use a constructor instead of `Create`, we'd improve runtime codegen, at the expense of narrowing the pattern a bit.

Answer: We will restrict to constructors for now. We can revisit adding a general `Create` method later if the

scenario arises.

Append... method overload resolution

Given an *applicable_interpolated_string_handler_type* `T` and an *interpolated_string_expression* `i`, overload resolution for a set of valid `Append...` methods on `T` is performed as follows:

1. If there are any *interpolated_regular_string_character* components in `i`:
 - a. Member lookup on `T` with the name `AppendLiteral` is performed. The resulting method group is called `M1`.
 - b. The argument list `A1` is constructed with one value parameter of type `string`.
 - c. Traditional method invocation resolution is performed with method group `M1` and argument list `A1`. For the purposes of method invocation final validation, the context of `M1` is treated as a *member_access* through an instance of `T`.
 - If a single-best method `Fi` is found and no errors were produced, the result of method invocation resolution is `Fi`.
 - Otherwise, an error is reported.
2. For every *interpolation* `ix` component of `i`:
 - a. Member lookup on `T` with the name `AppendFormatted` is performed. The resulting method group is called `Mf`.
 - b. The argument list `Af` is constructed:
 - a. The first parameter is the *expression* of `ix`, passed by value.
 - b. If `ix` directly contains a *constant_expression* component, then an integer value parameter is added, with the name `alignment` specified.
 - c. If `ix` is directly followed by an *interpolation_format*, then a string value parameter is added, with the name `format` specified.
 - c. Traditional method invocation resolution is performed with method group `Mf` and argument list `Af`. For the purposes of method invocation final validation, the context of `Mf` is treated as a *member_access* through an instance of `T`.
 - If a single-best method `Fi` is found, the result of method invocation resolution is `Fi`.
 - Otherwise, an error is reported.
3. Finally, for every `Fi` discovered in steps 1 and 2, final validation is performed:
 - If any `Fi` does not return `bool` by value or `void`, an error is reported.
 - If all `Fi` do not return the same type, an error is reported.

Note that these rules do not permit extension methods for the `Append...` calls. We could consider enabling that if we choose, but this is analogous to the enumerator pattern, where we allow `GetEnumerator` to be an extension method, but not `Current` or `MoveNext()`.

These rules *do* permit default parameters for the `Append...` calls, which will work with things like `CallerLineNumber` or `CallerArgumentExpression` (when supported by the language).

We have separate overload lookup rules for base elements vs interpolation holes because some handlers will want to be able to understand the difference between the components that were interpolated and the components that were part of the base string.

Open Question

Some scenarios, like structured logging, want to be able to provide names for interpolation elements. For example, today a logging call might look like `Log("{name} bought {itemCount} items", name, items.Count);`. The names inside the `{}` provide important structure information for loggers that help with ensuring output is consistent and uniform. Some cases might be able to reuse the `:format` component of an interpolation hole for this, but many loggers already understand format specifiers and have existing behavior for output formatting

based on this info. Is there some syntax we can use to enable putting these named specifiers in?

Some cases may be able to get away with `CallerArgumentExpression`, provided that support does land in C# 10. But for cases that invoke a method/property, that may not be sufficient.

Answer:

While there are some interesting parts to templated strings we could explore in an orthogonal language feature, we don't think a specific syntax here has much benefit over solutions such as using a tuple:

```
($"{("StructuredCategory", myExpression)})".
```

Performing the conversion

Given an *applicable_interpolated_string_handler_type* `T` and an *interpolated_string_expression* `i` that had a valid constructor `Fc` and `Append...` methods `Fa` resolved, lowering for `i` is performed as follows:

1. Any arguments to `Fc` that occur lexically before `i` are evaluated and stored into temporary variables in lexical order. In order to preserve lexical ordering, if `i` occurred as part of a larger expression `e`, any components of `e` that occurred before `i` will be evaluated as well, again in lexical order.
2. `Fc` is called with the length of the interpolated string literal components, the number of *interpolation* holes, any previously evaluated arguments, and a `bool` out argument (if `Fc` was resolved with one as the last parameter). The result is stored into a temporary value `ib`.
 - a. The length of the literal components is calculated after replacing any *open_brace_escape_sequence* with a single `{`, and any *close_brace_escape_sequence* with a single `}`.
3. If `Fc` ended with a `bool` out argument, a check on that `bool` value is generated. If true, the methods in `Fa` will be called. Otherwise, they will not be called.
4. For every `Fax` in `Fa`, `Fax` is called on `ib` with either the current literal component or *interpolation* expression, as appropriate. If `Fax` returns a `bool`, the result is logically anded with all preceding `Fax` calls.
 - a. If `Fax` is a call to `AppendLiteral`, the literal component is unescaped by replacing any *open_brace_escape_sequence* with a single `{`, and any *close_brace_escape_sequence* with a single `}`.
5. The result of the conversion is `ib`.

Again, note that arguments passed to `Fc` and arguments passed to `e` are the same temp. Conversions may occur on top of the temp to convert to a form that `Fc` requires, but for example lambdas cannot be bound to a different delegate type between `Fc` and `e`.

Open Question

This lowering means that subsequent parts of the interpolated string after a false-returning `Append...` call don't get evaluated. This could potentially be very confusing, particularly if the format hole is side-effecting. We could instead evaluate all format holes first, then repeatedly call `Append...` with the results, stopping if it returns false. This would ensure that all expressions get evaluated as one might expect, but we call as few methods as we need to. While the partial evaluation might be desirable for some more advanced cases, it is perhaps non-intuitive for the general case.

Another alternative, if we want to always evaluate all format holes, is to remove the `Append...` version of the API and just do repeated `Format` calls. The handler can track whether it should just be dropping the argument and immediately returning for this version.

Answer: We will have conditional evaluation of the holes.

Open Question

Do we need to dispose of disposable handler types, and wrap calls with try/finally to ensure that `Dispose` is called? For example, the interpolated string handler in the bcl might have a rented array inside it, and if one of

the interpolation holes throws an exception during evaluation, that rented array could be leaked if it wasn't disposed.

Answer: No. handlers can be assigned to locals (such as `MyHandler handler = $"{MyCode()}";`), and the lifetime of such handlers is unclear. Unlike foreach enumerators, where the lifetime is obvious and no user-defined local is created for the enumerator.

Other considerations

Allow `string` types to be convertible to handlers as well

For type author simplicity, we could consider allowing expressions of type `string` to be implicitly-convertible to *applicable_interpolated_string_handler_types*. As proposed today, authors will likely need to overload on both that handler type and regular `string` types, so their users don't have to understand the difference. This may be an annoying and non-obvious overhead, as a `string` expression can be viewed as an interpolation with `expression.Length` prefilled length and 0 holes to be filled.

This would allow new APIs to only expose a handler, without also having to expose a `string`-accepting overload. However, it won't get around the need for changes to better conversion from expression, so while it would work it may be unnecessary overhead.

Answer:

We think that this could end up being confusing, and there's an easy workaround for custom handler types: add a user-defined conversion from string.

Incorporating spans for heap-less strings

`ValueStringBuilder` as it exists today has 2 constructors: one that takes a count, and allocates on the heap eagerly, and one that takes a `Span<char>`. That `Span<char>` is usually a fixed size in the runtime codebase, around 250 elements on average. To truly replace that type, we should consider an extension to this where we also recognize `GetInterpolatedString` methods that take a `Span<char>`, instead of just the count version. However, we see a few potential thorny cases to resolve here:

- We don't want to `stackalloc` repeatedly in a hot loop. If we were to do this extension to the feature, we'd likely want to share the `stackalloc`'d span between loop iterations. We know this is safe, as `Span<T>` is a ref struct that can't be stored on the heap, and users would have to be pretty devious to manage to extract a reference to that `Span` (such as creating a method that accepts such a handler then deliberately retrieving the `Span` from the handler and returning it to the caller). However, allocating ahead of time produces other questions:
 - Should we eagerly `stackalloc`? What if the loop is never entered, or exits before it needs the space?
 - If we don't eagerly `stackalloc`, does that mean we introduce a hidden branch on every loop? Most loops likely won't care about this, but it could affect some tight loops that don't want to pay the cost.
- Some strings can be quite big, and the appropriate amount to `stackalloc` is dependent on a number of factors, including runtime factors. We don't really want the C# compiler and specification to have to determine this ahead of time, so we'd want to resolve <https://github.com/dotnet/runtime/issues/25423> and add an API for the compiler to call in these cases. It also adds more pros and cons to the points from the previous loop, where we don't want to potentially allocate large arrays on the heap many times or before one is needed.

Answer:

This is out of scope for C# 10. We can look at this in general when we look at the more general `params Span<T>` feature.

Non-try version of the API

For simplicity, this spec currently just proposes recognizing a `Append...` method, and things that always

succeed (like `InterpolatedStringHandler`) would always return true from the method. This was done to support partial formatting scenarios where the user wants to stop formatting if an error occurs or if it's unnecessary, such as the logging case, but could potentially introduce a bunch of unnecessary branches in standard interpolated string usage. We could consider an addendum where we use just `FormatX` methods if no `Append...` method is present, but it does present questions about what we do if there's a mix of both `Append...` and `FormatX` calls.

Answer:

We want the non-try version of the API. The proposal has been updated to reflect this.

Passing previous arguments to the handler

There is unfortunate lack of symmetry in the proposal at it currently exists: invoking an extension method in reduced form produces different semantics than invoking the extension method in normal form. This is different from most other locations in the language, where reduced form is just a sugar. We propose adding an attribute to the framework that we will recognize when binding a method, that informs the compiler that certain parameters should be passed to the constructor on the handler. Usage looks like this:

```
namespace System.Runtime.CompilerServices
{
    [AttributeUsage(AttributeTargets.Parameter, AllowMultiple = false, Inherited = false)]
    public sealed class InterpolatedStringHandlerArgumentAttribute : Attribute
    {
        public InterpolatedStringHandlerArgumentAttribute(string argument);
        public InterpolatedStringHandlerArgumentAttribute(params string[] arguments);

        public string[] Arguments { get; }
    }
}
```

Usage of this is then:

```

namespace System
{
    public sealed class String
    {
        public static string Format(IFormatProvider? provider,
[InterpolatedStringHandlerArgument("provider")] ref DefaultInterpolatedStringHandler handler);
        ...
    }
}

namespace System.Runtime.CompilerServices
{
    public ref struct DefaultInterpolatedStringHandler
    {
        public DefaultInterpolatedStringHandler(int baseLength, int holeCount, IFormatProvider? provider);
// additional factory
        ...
    }
}

var formatted = string.Format(CultureInfo.InvariantCulture, $"{X} = {Y}");

// Is lowered to

var tmp1 = CultureInfo.InvariantCulture;
var handler = new DefaultInterpolatedStringHandler(3, 2, tmp1);
handler.AppendFormatted(X);
handler.AppendLiteral(" = ");
handler.AppendFormatted(Y);
var formatted = string.Format(tmp1, handler);

```

The questions we need to answer:

1. Do we like this pattern in general?
2. Do we want to allow these arguments to come from after the handler parameter? Some existing patterns in the BCL, such as `Utf8Formatter`, put the value to be formatted *before* the thing needed to format into. To fit in best with these patterns, we'd likely want to allow this, but we need to decide if this out-of-order evaluate is ok.

Answer:

We want to support this. The spec has been updated to reflect this. Arguments will be required to be specified in lexical order at the call site, and if a needed argument to the create method is specified after the interpolated string literal, an error is produced.

`await` usage in interpolation holes

Because `($"{await A()}")` is a valid expression today, we need to rationalize how interpolation holes with `await`. We could solve this with a few rules:

1. If an interpolated string used as a `string`, `IFormattable`, or `FormattableString` has an `await` in an interpolation hole, fall back to old-style formatter.
2. If an interpolated string is subject to an *implicit_string_handler_conversion* and *applicable_interpolated_string_handler_type* is a `ref struct`, `await` is not allowed to be used in the format holes.

Fundamentally, this desugaring could use a `ref struct` in an async method as long as we guarantee that the `ref struct` will not need to be saved to the heap, which should be possible if we forbid `await`s in the interpolation holes.

Alternatively, we could simply make all handler types non-ref structs, including the framework handler for

interpolated strings. This would, however, preclude us from someday recognizing a `Span` version that does not need to allocate any scratch space at all.

Answer:

We will treat interpolated string handlers the same as any other type: this means that if the handler type is a `ref struct` and the current context doesn't allow the usage of `ref structs`, it is illegal to use handler here. The spec around lowering of string literals used as strings is intentionally vague to allow the compiler to decide on what rules it deems appropriate, but for custom handler types they will have to follow the same rules as the rest of the language.

Handlers as ref parameters

Some handlers might want to be passed as `ref` parameters (either `in` or `ref`). Should we allow either? And if so, what will a `ref` handler look like? `ref $" "` is confusing, as you're not actually passing the string by `ref`, you're passing the handler that is created from the `ref` by `ref`, and has similar potential issues with `async` methods.

Answer:

We want to support this. The spec has been updated to reflect this. The rules should reflect the same rules that apply to extension methods on value types.

Interpolated strings through binary expressions and conversions

Because this proposal makes interpolated strings context sensitive, we would like to allow the compiler to treat a binary expression composed entirely of interpolated strings, or an interpolated string subjected to a cast, as an interpolated string literal for the purposes of overload resolution. For example, take the following scenario:

```
struct Handler1
{
    public Handler1(int literalLength, int formattedCount, C c) => ...;
    // AppendX... methods as necessary
}
struct Handler2
{
    public Handler2(int literalLength, int formattedCount, C c) => ...;
    // AppendX... methods as necessary
}

class C
{
    void M(Handler1 handler) => ...;
    void M(Handler2 handler) => ...;
}

c.M($"X"); // Ambiguous between the M overloads
```

This would be ambiguous, necessitating a cast to either `Handler1` or `Handler2` in order to resolve. However, in making that cast, we would potentially throw away the information that there is context from the method receiver, meaning that the cast would fail because there is nothing to fill in the information of `c`. A similar issue arises with binary concatenation of strings: the user could want to format the literal across several lines to avoid line wrapping, but would not be able to because that would no longer be an interpolated string literal convertible to the handler type.

To resolve these cases, we make the following changes:

- An *additive_expression* composed entirely of *interpolated_string_expressions* and using only `+` operators is considered to be an *interpolated_string_literal* for the purposes of conversions and overload resolution. The final interpolated string is created by logically concatenating all individual *interpolated_string_expression* components, from left to right.

- A *cast_expression* or a *relational_expression* with operator `as` whose operand is an *interpolated_string_expressions* is considered an *interpolated_string_expressions* for the purposes of conversions and overload resolution.

Open Questions:

Do we want to do this? We don't do this for `System.FormattableString`, for example, but that can be broken out onto a different line, whereas this can be context-dependent and therefore not able to be broken out into a different line. There are also no overload resolution concerns with `FormattableString` and `IFormattable`.

Answer.

We think that this is a valid use case for additive expressions, but that the cast version is not compelling enough at this time. We can add it later if necessary. The spec has been updated to reflect this decision.

Other use cases

See <https://github.com/dotnet/runtime/issues/50635> for examples of proposed handler APIs using this pattern.

Constant Interpolated Strings

12/28/2021 • 3 minutes to read • [Edit Online](#)

Summary

Enables constants to be generated from interpolated strings of type string constant.

Motivation

The following code is already legal:

```
public class C
{
    const string S1 = "Hello world";
    const string S2 = "Hello" + " " + "World";
    const string S3 = S1 + " Kevin, welcome to the team!";
}
```

However, there have been many community requests to make the following also legal:

```
public class C
{
    const string S1 = $"Hello world";
    const string S2 = $"Hello{" " }World";
    const string S3 = $"{S1} Kevin, welcome to the team!";
}
```

This proposal represents the next logical step for constant string generation, where existing string syntax that works in other situations is made to work for constants.

Detailed design

The following represent the updated specifications for constant expressions under this new proposal. Current specifications from which this was directly based on can be found [here](#).

Constant Expressions

A *constant_expression* is an expression that can be fully evaluated at compile-time.

```
constant_expression
: expression
;
```

A constant expression must be the `null` literal or a value with one of the following types: `sbyte`, `byte`, `short`, `ushort`, `int`, `uint`, `long`, `ulong`, `char`, `float`, `double`, `decimal`, `bool`, `object`, `string`, or any enumeration type. Only the following constructs are permitted in constant expressions:

- Literals (including the `null` literal).
- References to `const` members of class and struct types.
- References to members of enumeration types.
- References to `const` parameters or local variables
- Parenthesized sub-expressions, which are themselves constant expressions.

- Cast expressions, provided the target type is one of the types listed above.
- `checked` and `unchecked` expressions
- Default value expressions
- Nameof expressions
- The predefined `+`, `-`, `!`, and `~` unary operators.
- The predefined `+`, `-`, `*`, `/`, `%`, `<<`, `>>`, `&`, `|`, `^`, `&&`, `||`, `==`, `!=`, `<`, `>`, `<=`, and `>=` binary operators, provided each operand is of a type listed above.
- The `?:` conditional operator.
- *Interpolated strings `{ }`, provided that all components are constant expressions of type `string` and all interpolated components lack alignment and format specifiers.*

The following conversions are permitted in constant expressions:

- Identity conversions
- Numeric conversions
- Enumeration conversions
- Constant expression conversions
- Implicit and explicit reference conversions, provided that the source of the conversions is a constant expression that evaluates to the null value.

Other conversions including boxing, unboxing and implicit reference conversions of non-null values are not permitted in constant expressions. For example:

```
class C
{
    const object i = 5;           // error: boxing conversion not permitted
    const object str = "hello"; // error: implicit reference conversion
}
```

the initialization of `i` is an error because a boxing conversion is required. The initialization of `str` is an error because an implicit reference conversion from a non-null value is required.

Whenever an expression fulfills the requirements listed above, the expression is evaluated at compile-time. This is true even if the expression is a sub-expression of a larger expression that contains non-constant constructs.

The compile-time evaluation of constant expressions uses the same rules as run-time evaluation of non-constant expressions, except that where run-time evaluation would have thrown an exception, compile-time evaluation causes a compile-time error to occur.

Unless a constant expression is explicitly placed in an `unchecked` context, overflows that occur in integral-type arithmetic operations and conversions during the compile-time evaluation of the expression always cause compile-time errors ([Constant expressions](#)).

Constant expressions occur in the contexts listed below. In these contexts, a compile-time error occurs if an expression cannot be fully evaluated at compile-time.

- Constant declarations ([Constants](#)).
- Enumeration member declarations ([Enum members](#)).
- Default arguments of formal parameter lists ([Method parameters](#))
- `case` labels of a `switch` statement ([The switch statement](#)).
- `goto case` statements ([The goto statement](#)).
- Dimension lengths in an array creation expression ([Array creation expressions](#)) that includes an initializer.
- Attributes ([Attributes](#)).

An implicit constant expression conversion ([Implicit constant expression conversions](#)) permits a constant expression of type `int` to be converted to `sbyte`, `byte`, `short`, `ushort`, `uint`, or `ulong`, provided the value of the constant expression is within the range of the destination type.

Drawbacks

This proposal adds additional complexity to the compiler in exchange for broader applicability of interpolated strings. As these strings are fully evaluated at compile time, the valuable automatic formatting features of interpolated strings are less necessary. Most use cases can be largely replicated through the alternatives below.

Alternatives

The current `+` operator for string concatenation can combine strings in a similar manner to the current proposal.

Unresolved questions

What parts of the design are still undecided?

Design meetings

Link to design notes that affect this proposal, and describe in one sentence for each what changes they led to.

Lambda improvements

12/28/2021 • 13 minutes to read • [Edit Online](#)

Summary

Proposed changes:

1. Allow lambdas with attributes
2. Allow lambdas with explicit return type
3. Infer a natural delegate type for lambdas and method groups

Motivation

Support for attributes on lambdas would provide parity with methods and local functions.

Support for explicit return types would provide symmetry with lambda parameters where explicit types can be specified. Allowing explicit return types would also provide control over compiler performance in nested lambdas where overload resolution must bind the lambda body currently to determine the signature.

A natural type for lambda expressions and method groups will allow more scenarios where lambdas and method groups may be used without an explicit delegate type, including as initializers in `var` declarations.

Requiring explicit delegate types for lambdas and method groups has been a friction point for customers, and has become an impediment to progress in ASP.NET with recent work on [MapAction](#).

[ASP.NET MapAction](#) without proposed changes (`MapAction()` takes a `System.Delegate` argument):

```
[HttpGet("/")] Todo GetTodo() => new(Id: 0, Name: "Name");
app.MapAction((Func<Todo>)GetTodo);

[HttpPost("/")] Todo PostTodo([FromBody] Todo todo) => todo;
app.MapAction((Func<Todo, Todo>)PostTodo);
```

[ASP.NET MapAction](#) with natural types for method groups:

```
[HttpGet("/")] Todo GetTodo() => new(Id: 0, Name: "Name");
app.MapAction(GetTodo);

[HttpPost("/")] Todo PostTodo([FromBody] Todo todo) => todo;
app.MapAction(PostTodo);
```

[ASP.NET MapAction](#) with attributes and natural types for lambda expressions:

```
app.MapAction([HttpGet("/")] () => new Todo(Id: 0, Name: "Name"));
app.MapAction([HttpPost("/")] ([FromBody] Todo todo) => todo);
```

Attributes

Attributes may be added to lambda expressions and lambda parameters. To avoid ambiguity between method attributes and parameter attributes, a lambda expression with attributes must use a parenthesized parameter list. Parameter types are not required.

```
f = [A] () => { };           // [A] lambda
f = [return:A] x => x;        // syntax error at '>'
f = [return:A] (x) => x;      // [A] lambda
f = [A] static x => x;        // syntax error at '>'

f = ([A] x) => x;             // [A] x
f = ([A] ref int x) => x;     // [A] x
```

Multiple attributes may be specified, either comma-separated within the same attribute list or as separate attribute lists.

```
var f = [A1, A2][A3] () => { };    // ok
var g = ([A1][A2, A3] int x) => x; // ok
```

Attributes are not supported for *anonymous methods* declared with `delegate { }` syntax.

```
f = [A] delegate { return 1; };      // syntax error at 'delegate'
f = delegate ([A] int x) { return x; }; // syntax error at '['
```

The parser will look ahead to differentiate a collection initializer with an element assignment from a collection initializer with a lambda expression.

```
var y = new C { [A] = x };    // ok: y[A] = x
var z = new C { [A] x => x }; // ok: z[0] = [A] x => x
```

The parser will treat `?[` as the start of a conditional element access.

```
x = b ? [A];                // ok
y = b ? [A] () => { } : z;    // syntax error at '('
```

Attributes on the lambda expression or lambda parameters will be emitted to metadata on the method that maps to the lambda.

In general, customers should not depend on how lambda expressions and local functions map from source to metadata. How lambdas and local functions are emitted can, and has, changed between compiler versions.

The changes proposed here are targeted at the `Delegate` driven scenario. It should be valid to inspect the `MethodInfo` associated with a `Delegate` instance to determine the signature of the lambda expression or local function including any explicit attributes and additional metadata emitted by the compiler such as default parameters. This allows teams such as ASP.NET to make available the same behaviors for lambdas and local functions as ordinary methods.

Explicit return type

An explicit return type may be specified before the parenthesized parameter list.

```
f = T () => default;          // ok
f = short x => 1;              // syntax error at '>'
f = ref int (ref int x) => ref x; // ok
f = static void (_) => { };      // ok
f = async async (async async) => async; // ok?
```

The parser will look ahead to differentiate a method call `T()` from a lambda expression `T () => e`.

Explicit return types are not supported for anonymous methods declared with `delegate { }` syntax.

```
f = delegate int { return 1; };           // syntax error
f = delegate int (int x) { return x; };   // syntax error
```

Method type inference should make an exact inference from an explicit lambda return type.

```
static void F<T>(Func<T, T> f) { ... }
F(int (i) => i); // Func<int, int>
```

Variance conversions are not allowed from lambda return type to delegate return type (matching similar behavior for parameter types).

```
Func<object> f1 = string () => null; // error
Func<object?> f2 = object () => x;   // warning
```

The parser allows lambda expressions with `ref` return types within expressions without additional parentheses.

```
d = ref int () => x; // d = (ref int () => x)
F(ref int () => x); // F((ref int () => x))
```

`var` cannot be used as an explicit return type for lambda expressions.

```
class var { }

d = var (var v) => v;           // error: contextual keyword 'var' cannot be used as explicit lambda
return type
d = @var (var v) => v;          // ok
d = ref var (ref var v) => ref v; // error: contextual keyword 'var' cannot be used as explicit lambda
return type
d = ref @var (ref var v) => ref v; // ok
```

Natural (function) type

An *anonymous function expression* (a *lambda expression* or an *anonymous method*) has a natural type if the parameters types are explicit and the return type is either explicit or can be inferred (see [inferred return type](#)).

A *method group* has a natural type if all candidate methods in the method group have a common signature. (If the method group may include extension methods, the candidates include the containing type and all extension method scopes.)

The natural type of an anonymous function expression or method group is a *function_type*. A *function_type* represents a method signature: the parameter types and ref kinds, and return type and ref kind. Anonymous function expressions or method groups with the same signature have the same *function_type*.

Function_types are used in a few specific contexts only:

- implicit and explicit conversions
- [method type inference](#) and [best common type](#)
- `var` initializers

A *function_type* exists at compile time only: *function_types* do not appear in source or metadata.

Conversions

From a *function_type* `F` there are implicit *function_type* conversions:

- To a *function_type* `G` if the parameters and return types of `F` are variance-convertible to the parameters and return type of `G`
- To `System.MulticastDelegate` or base classes or interfaces of `System.MulticastDelegate`
- To `System.Linq.Expressions.Expression` or `System.Linq.Expressions.LambdaExpression`

Anonymous function expressions and method groups already have *conversions from expression* to delegate types and expression tree types (see [anonymous function conversions](#) and [method group conversions](#)). Those conversions are sufficient for converting to strongly-typed delegate types and expression tree types. The *function_type* conversions above add *conversions from type* to the base types only: `System.MulticastDelegate`, `System.Linq.Expressions.Expression`, etc.

There are no conversions to a *function_type* from a type other than a *function_type*. There are no explicit conversions for *function_types* since *function_types* cannot be referenced in source.

A conversion to `System.MulticastDelegate` or base type or interface realizes the anonymous function or method group as an instance of an appropriate delegate type. A conversion to `System.Linq.Expressions.Expression<TDelegate>` or base type realizes the lambda expression as an expression tree with an appropriate delegate type.

```
Delegate d = delegate (object obj) { }; // Action<object>
Expression e = () => "";                // Expression<Func<string>>
object o = "".Clone;                   // Func<object>
```

Function_type conversions are not implicit or explicit [standard conversions](#) and are not considered when determining whether a user-defined conversion operator is applicable to an anonymous function or method group. From [evaluation of user defined conversions](#):

For a conversion operator to be applicable, it must be possible to perform a standard conversion ([Standard conversions](#)) from the source type to the operand type of the operator, and it must be possible to perform a standard conversion from the result type of the operator to the target type.

```
class C
{
    public static implicit operator C(Delegate d) { ... }
}

C c;
c = () => 1;      // error: cannot convert lambda expression to type 'C'
c = (C) (() => 2); // error: cannot convert lambda expression to type 'C'
```

A warning is reported for an implicit conversion of a method group to `object`, since the conversion is valid but perhaps unintentional.

```
Random r = new Random();
object obj;
obj = r.NextDouble;           // warning: Converting method group to 'object'. Did you intend to invoke the method?
obj = (object)r.NextDouble; // ok
```

Type inference

The existing rules for type inference are mostly unchanged (see [type inference](#)). There are however a **couple of changes** below to specific phases of type inference.

First phase

The [first phase](#) allows an anonymous function to bind to T_i even if T_i is not a delegate or expression tree type (perhaps a type parameter constrained to `System.Delegate` for instance).

For each of the method arguments E_i :

- If E_i is an anonymous function and T_i is a delegate type or expression tree type, an *explicit parameter type inference* is made from E_i to T_i and an *explicit return type inference* is made from E_i to T_i .
- Otherwise, if E_i has a type U and x_i is a value parameter then a *lower-bound inference* is made from U to T_i .
- Otherwise, if E_i has a type U and x_i is a `ref` or `out` parameter then an *exact inference* is made from U to T_i .
- Otherwise, no inference is made for this argument.

Explicit return type inference

An *explicit return type inference* is made from an expression E to a type T in the following way:

- If E is an anonymous function with explicit return type U_r and T is a delegate type or expression tree type with return type V_r then an *exact inference* ([Exact inferences](#)) is made from U_r to V_r .

Fixing

[Fixing](#) ensures other conversions are preferred over *function_type* conversions. (Lambda expressions and method group expressions only contribute to lower bounds so handling of *function_types* is needed for lower bounds only.)

An *unfixed* type variable x_i with a set of bounds is *fixed* as follows:

- The set of *candidate types* u_j starts out as the set of all types in the set of bounds for x_i where **function types are ignored in lower bounds if there any types that are not function types**.
- We then examine each bound for x_i in turn: For each exact bound u of x_i all types u_j which are not identical to u are removed from the candidate set. For each lower bound u of x_i all types u_j to which there is *not* an implicit conversion from u are removed from the candidate set. For each upper bound u of x_i all types u_j from which there is *not* an implicit conversion to u are removed from the candidate set.
- If among the remaining candidate types u_j there is a unique type v from which there is an implicit conversion to all the other candidate types, then x_i is fixed to v .
- Otherwise, type inference fails.

Best common type

[Best common type](#) is defined in terms of type inference so the type inference changes above apply to best common type as well.

```
var fs = new[] { (string s) => s.Length; (string s) => int.Parse(s) } // Func<string, int>[]
```

`var`

Anonymous functions and method groups with function types can be used as initializers in `var` declarations.


```

var f1 = () => default;           // error: cannot infer type
var f2 = x => x;                  // error: cannot infer type
var f3 = () => 1;                 // System.Func<int>
var f4 = string () => null;       // System.Func<string>
var f5 = delegate (object o) { }; // System.Action<object>

static void F1() { }
static void F1<T>(this T t) { }
static void F2(this string s) { }

var f6 = F1;    // error: multiple methods
var f7 = "", F1; // System.Action
var f8 = F2;    // System.Action<string>

```

Function types are not used in assignments to discards.

```

d = () => 0; // ok
_ = () => 1; // error

```

Delegate types

The delegate type for the anonymous function or method group with parameter types `P1, ..., Pn` and return type `R` is:

- if any parameter or return value is not by value, or there are more than 16 parameters, or any of the parameter types or return are not valid type arguments (say, `(int* p) => { }`), then the delegate is a synthesized `internal` anonymous delegate type with signature that matches the anonymous function or method group, and with parameter names `arg1, ..., argn` or `arg` if a single parameter;
- if `R` is `void`, then the delegate type is `System.Action<P1, ..., Pn>`;
- otherwise the delegate type is `System.Func<P1, ..., Pn, R>`.

The compiler may allow more signatures to bind to `System.Action<>` and `System.Func<>` types in the future (if `ref struct` types are allowed type arguments for instance).

`modopt()` or `modreq()` in the method group signature are ignored in the corresponding delegate type.

If two anonymous functions or method groups in the same compilation require synthesized delegate types with the same parameter types and modifiers and the same return type and modifiers, the compiler will use the same synthesized delegate type.

Overload resolution

[Better function member](#) is updated to prefer members where none of the conversions and none of the type arguments involved inferred types from lambda expressions or method groups.

Better function member

... Given an argument list `A` with a set of argument expressions `{E1, E2, ..., En}` and two applicable function members `Mp` and `Mq` with parameter types `{P1, P2, ..., Pn}` and `{Q1, Q2, ..., Qn}`, `Mp` is defined to be a **better function member** than `Mq` if

1. for each argument, the implicit conversion from `Ex` to `Px` is not a *function_type_conversion*, and
 - `Mp` is a non-generic method or `Mp` is a generic method with type parameters `{X1, X2, ..., Xp}` and for each type parameter `Xi` the type argument is inferred from an expression or from a type other than a *function_type*, and
 - for at least one argument, the implicit conversion from `Ex` to `Qx` is a *function_type_conversion*, or `Mq` is a generic method with type parameters

`{Y1, Y2, ..., Yq}` and for at least one type parameter `Yi` the type argument is inferred from a *function_type*, or

- for each argument, the implicit conversion from `Ex` to `Qx` is not better than the implicit conversion from `Ex` to `Px`, and for at least one argument, the conversion from `Ex` to `Px` is better than the conversion from `Ex` to `Qx`.

[Better conversion from expression](#) is updated to prefer conversions that did not involve inferred types from lambda expressions or method groups.

Better conversion from expression

Given an implicit conversion `c1` that converts from an expression `E` to a type `T1`, and an implicit conversion `c2` that converts from an expression `E` to a type `T2`, `c1` is a *better conversion* than `c2` if:

- `c1` is not a *function_type_conversion* and `c2` is a *function_type_conversion*, or
- `E` is a non-constant *interpolated_string_expression*, `c1` is an *implicit_string_handler_conversion*, `T1` is an *applicable_interpolated_string_handler_type*, and `c2` is not an *implicit_string_handler_conversion*, or
- `E` does not exactly match `T2` and at least one of the following holds:
 - `E` exactly matches `T1` ([Exactly matching Expression](#))
 - `T1` is a better conversion target than `T2` ([Better conversion target](#))

Syntax

```
lambda_expression
: modifier* identifier '=>' (block | expression)
| attribute_list* modifier* type? lambda_parameters '=>' (block | expression)
;

lambda_parameters
: lambda_parameter
| '(' (lambda_parameter (',' lambda_parameter)*)? ')'
;

lambda_parameter
: identifier
| attribute_list* modifier* type? identifier equals_value_clause?
;
```

Open issues

Should default values be supported for lambda expression parameters for completeness?

Should `System.Diagnostics.ConditionalAttribute` be disallowed on lambda expressions since there are few scenarios where a lambda expression could be used conditionally?

```
([Conditional("DEBUG")] static (x, y) => Assert(x == y))(a, b); // ok?
```

Should the *function_type* be available from the compiler API, in addition to the resulting delegate type?

Currently, the inferred delegate type uses `System.Action<>` or `System.Func<>` when parameter and return types are valid type arguments *and* there are no more than 16 parameters, and if the expected `Action<>` or `Func<>` type is missing, an error is reported. Instead, should the compiler use `System.Action<>` or `System.Func<>` regardless of arity? And if the expected type is missing, synthesize a delegate type otherwise?

CallerArgumentExpression

12/28/2021 • 7 minutes to read • [Edit Online](#)

Summary

Allow developers to capture the expressions passed to a method, to enable better error messages in diagnostic/testing APIs and reduce keystrokes.

Motivation

When an assertion or argument validation fails, the developer wants to know as much as possible about where and why it failed. However, today's diagnostic APIs do not fully facilitate this. Consider the following method:

```
T Single<T>(this T[] array)
{
    Debug.Assert(array != null);
    Debug.Assert(array.Length == 1);

    return array[0];
}
```

When one of the asserts fail, only the filename, line number, and method name will be provided in the stack trace. The developer will not be able to tell which assert failed from this information-- (s)he will have to open the file and navigate to the provided line number to see what went wrong.

This is also the reason testing frameworks have to provide a variety of assert methods. With xUnit, `Assert.True` and `Assert.False` are not frequently used because they do not provide enough context about what failed.

While the situation is a bit better for argument validation because the names of invalid arguments are shown to the developer, the developer must pass these names to exceptions manually. If the above example were rewritten to use traditional argument validation instead of `Debug.Assert`, it would look like

```
T Single<T>(this T[] array)
{
    if (array == null)
    {
        throw new ArgumentNullException(nameof(array));
    }

    if (array.Length != 1)
    {
        throw new ArgumentException("Array must contain a single element.", nameof(array));
    }

    return array[0];
}
```

Notice that `nameof(array)` must be passed to each exception, although it's already clear from context which argument is invalid.

Detailed design

In the above examples, including the string `"array != null"` or `"array.Length == 1"` in the assert message

would help the developer determine what failed. Enter `CallerArgumentExpression`: it's an attribute the framework can use to obtain the string associated with a particular method argument. We would add it to `Debug.Assert` like so

```
public static class Debug
{
    public static void Assert(bool condition, [CallerArgumentExpression("condition")] string message = null);
}
```

The source code in the above example would stay the same. However, the code the compiler actually emits would correspond to

```
T Single<T>(this T[] array)
{
    Debug.Assert(array != null, "array != null");
    Debug.Assert(array.Length == 1, "array.Length == 1");

    return array[0];
}
```

The compiler specially recognizes the attribute on `Debug.Assert`. It passes the string associated with the argument referred to in the attribute's constructor (in this case, `condition`) at the call site. When either assert fails, the developer will be shown the condition that was false and will know which one failed.

For argument validation, the attribute cannot be used directly, but can be made use of through a helper class:

```

public static class Verify
{
    public static void Argument(bool condition, string message, [CallerArgumentExpression("condition")]
string conditionExpression = null)
    {
        if (!condition) throw new ArgumentException(message: message, paramName: conditionExpression);
    }

    public static void InRange(int argument, int low, int high,
        [CallerArgumentExpression("argument")] string argumentExpression = null,
        [CallerArgumentExpression("low")] string lowExpression = null,
        [CallerArgumentExpression("high")] string highExpression = null)
    {
        if (argument < low)
        {
            throw new ArgumentOutOfRangeException(paramName: argumentExpression,
                message: $"{argumentExpression} ({argument}) cannot be less than {lowExpression} ({low}).");
        }

        if (argument > high)
        {
            throw new ArgumentOutOfRangeException(paramName: argumentExpression,
                message: $"{argumentExpression} ({argument}) cannot be greater than {highExpression}
({high}).");
        }
    }

    public static void NotNull<T>(T argument, [CallerArgumentExpression("argument")] string
argumentExpression = null)
        where T : class
    {
        if (argument == null) throw new ArgumentNullException(paramName: argumentExpression);
    }
}

T Single<T>(this T[] array)
{
    Verify.NotNull(array); // paramName: "array"
    Verify.Argument(array.Length == 1, "Array must contain a single element."); // paramName: "array.Length
== 1"

    return array[0];
}

T ElementAt(this T[] array, int index)
{
    Verify.NotNull(array); // paramName: "array"
    // paramName: "index"
    // message: "index (-1) cannot be less than 0 (0).", or
    // "index (6) cannot be greater than array.Length - 1 (5)."
    Verify.InRange(index, 0, array.Length - 1);

    return array[index];
}

```

A proposal to add such a helper class to the framework is underway at <https://github.com/dotnet/corefx/issues/17068>. If this language feature was implemented, the proposal could be updated to take advantage of this feature.

Extension methods

The `this` parameter in an extension method may be referenced by `CallerArgumentExpression`. For example:

```
public static void ShouldBe<T>(this T @this, T expected, [CallerArgumentExpression("this")] string
thisExpression = null) {}
```

```
contestant.Points.ShouldBe(1337); // thisExpression: "contestant.Points"
```

`thisExpression` will receive the expression corresponding to the object before the dot. If it's called with static method syntax, e.g. `Ext.ShouldBe(contestant.Points, 1337)`, it will behave as if first parameter wasn't marked `this`.

There should always be an expression corresponding to the `this` parameter. Even if an instance of a class calls an extension method on itself, e.g. `this.Single()` from inside a collection type, the `this` is mandated by the compiler so `"this"` will get passed. If this rule is changed in the future, we can consider passing `null` or the empty string.

Extra details

- Like the other `Caller*` attributes, such as `CallerMemberName`, this attribute may only be used on parameters with default values.
- Multiple parameters marked with `CallerArgumentExpression` are permitted, as shown above.
- The attribute's namespace will be `System.Runtime.CompilerServices`.
- If `null` or a string that is not a parameter name (e.g. `"notAParameterName"`) is provided, the compiler will pass in an empty string.
- The type of the parameter `CallerArgumentExpressionAttribute` is applied to must have a standard conversion from `string`. This means no user-defined conversions from `string` are allowed, and in practice means the type of such a parameter must be `string`, `object`, or an interface implemented by `string`.

Drawbacks

- People who know how to use decompilers will be able to see some of the source code at call sites for methods marked with this attribute. This may be undesirable/unexpected for closed-source software.
- Although this is not a flaw in the feature itself, a source of concern may be that there exists a `Debug.Assert` API today that only takes a `bool`. Even if the overload taking a message had its second parameter marked with this attribute and made optional, the compiler would still pick the no-message one in overload resolution. Therefore, the no-message overload would have to be removed to take advantage of this feature, which would be a binary (although not source) breaking change.

Alternatives

- If being able to see source code at call sites for methods that use this attribute proves to be a problem, we can make the attribute's effects opt-in. Developers will enable it through an assembly-wide `[assembly: EnableCallerArgumentExpression]` attribute they put in `AssemblyInfo.cs`.
 - In the case the attribute's effects are not enabled, calling methods marked with the attribute would not be an error, to allow existing methods to use the attribute and maintain source compatibility. However, the attribute would be ignored and the method would be called with whatever default value was provided.

```

// Assembly1

void Foo(string bar); // V1
void Foo(string bar, string barExpression = "not provided"); // V2
void Foo(string bar, [CallerArgumentExpression("bar")] string barExpression = "not provided"); // V3

// Assembly2

Foo(a); // V1: Compiles to Foo(a), V2, V3: Compiles to Foo(a, "not provided")
Foo(a, "provided"); // V2, V3: Compiles to Foo(a, "provided")

// Assembly3

[assembly: EnableCallerArgumentExpression]

Foo(a); // V1: Compiles to Foo(a), V2: Compiles to Foo(a, "not provided"), V3: Compiles to Foo(a, "a")
Foo(a, "provided"); // V2, V3: Compiles to Foo(a, "provided")

```

- To prevent the [binary compatibility problem](#) from occurring every time we want to add new caller info to `Debug.Assert`, an alternative solution would be to add a `CallerInfo` struct to the framework that contains all the necessary information about the caller.

```

struct CallerInfo
{
    public string MemberName { get; set; }
    public string TypeName { get; set; }
    public string Namespace { get; set; }
    public string FullTypeName { get; set; }
    public string FilePath { get; set; }
    public int LineNumber { get; set; }
    public int ColumnNumber { get; set; }
    public Type Type { get; set; }
    public MethodBase Method { get; set; }
    public string[] ArgumentExpressions { get; set; }
}

[Flags]
enum CallerInfoOptions
{
    MemberName = 1, TypeName = 2, ...
}

public static class Debug
{
    public static void Assert(bool condition,
        // If a flag is not set here, the corresponding CallerInfo member is not populated by the caller, so
        // it's pay-for-play friendly.
        [CallerInfo(CallerInfoOptions.FilePath | CallerInfoOptions.Method |
        CallerInfoOptions.ArgumentExpressions)] CallerInfo callerInfo = default(CallerInfo))
    {
        string filePath = callerInfo.FilePath;
        MethodBase method = callerInfo.Method;
        string conditionExpression = callerInfo.ArgumentExpressions[0];
        ...
    }
}

class Bar
{
    void Foo()
    {
        Debug.Assert(false);

        // Translates to:

        var callerInfo = new CallerInfo();
        callerInfo.FilePath = @"C:\Bar.cs";
        callerInfo.Method = MethodBase.GetCurrentMethod();
        callerInfo.ArgumentExpressions = new string[] { "false" };
        Debug.Assert(false, callerInfo);
    }
}

```

This was originally proposed at <https://github.com/dotnet/csharp-lang/issues/87>.

There are a few disadvantages of this approach:

- Despite being pay-for-play friendly by allowing you to specify which properties you need, it could still hurt perf significantly by allocating an array for the expressions/calling `MethodBase.GetCurrentMethod` even when the assert passes.
- Additionally, while passing a new flag to the `CallerInfo` attribute won't be a breaking change, `Debug.Assert` won't be guaranteed to actually receive that new parameter from call sites that compiled against an old version of the method.

Unresolved questions

TBD

Design meetings

N/A

Enhanced #line directives

12/28/2021 • 8 minutes to read • [Edit Online](#)

Summary

The compiler applies the mapping defined by `#line` directives to diagnostic locations and sequence points emitted to the PDB.

Currently only the line number and file path can be mapped while the starting character is inferred from the source code. The proposal is to allow specifying full span mapping.

Motivation

DSLs that generate C# source code (such as ASPNET Razor) can't currently produce precise source mapping using `#line` directives. This results in degraded debugging experience in some cases as the sequence points emitted to the PDB can't map to the precise location in the original source code.

For example, the following Razor code

```
@page "/"
Time: @DateTime.Now
```

generates code like so (simplified):

```
#line hidden
void Render()
{
    _builder.Add("Time:");
    #line 2 "page.razor"
    _builder.Add(DateTime.Now);
    #line hidden
}
```

The above directive would map the sequence point emitted by the compiler for the `_builder.Add(DateTime.Now);` statement to the line 2, but the column would be off (16 instead of 7).

The Razor source generator actually incorrectly generates the following code:

```
#line hidden
void Render()
{
    _builder.Add("Time:");
    _builder.Add(
    #line 2 "page.razor"
        DateTime.Now
    #line hidden
    );
}
```

The intent was to preserve the starting character and it works for diagnostic location mapping. However, this does not work for sequence points since `#line` directive only applies to the sequence points that follow it. There is no sequence point in the middle of the `_builder.Add(DateTime.Now);` statement (sequence points can only be emitted at IL instructions with empty evaluation stack). The `#line 2` directive in above code thus has no

effect on the generated PDB and the debugger won't place a breakpoint or stop on the `@DateTime.Now` snippet in the Razor page.

Issues addressed by this proposal: <https://github.com/dotnet/roslyn/issues/43432>
<https://github.com/dotnet/roslyn/issues/46526>

Detailed design

We amend the syntax of `line_indicator` used in `pp_line` directive like so:

Current:

```
line_indicator
: decimal_digit+ whitespace file_name
| decimal_digit+
| 'default'
| 'hidden'
;
```

Proposed:

```
line_indicator
: '(' decimal_digit+ ',' decimal_digit+ ')' '-' '(' decimal_digit+ ',' decimal_digit+ ')' decimal_digit+
whitespace file_name
| '(' decimal_digit+ ',' decimal_digit+ ')' '-' '(' decimal_digit+ ',' decimal_digit+ ')' file_name
| decimal_digit+ whitespace file_name
| decimal_digit+
| 'default'
| 'hidden'
;
```

That is, the `#line` directive would accept either 5 decimal numbers (*start line, start character, end line, end character, character offset*), 4 decimal numbers (*start line, start character, end line, end character*), or a single one (*line*).

If *character offset* is not specified its default value is 0, otherwise it specifies the number of UTF-16 characters. The number must be non-negative and less than length of the line following the `#line` directive in the unmapped file.

(*start line, start character*)-(*end line, end character*) specifies a span in the mapped file. *start line* and *end line* are positive integers that specify line numbers. *start character, end character* are positive integers that specify UTF-16 character numbers. *start line, start character, end line, end character* are 1-based, meaning that the first line of the file and the first UTF-16 character on each line is assigned number 1.

The implementation would constraint these numbers so that they specify a valid [sequence point source span](#):

- *start line* - 1 is within range [0, 0x20000000) and not equal to 0xfeefee.
- *end line* - 1 is within range [0, 0x20000000) and not equal to 0xfeefee.
- *start character* - 1 is within range [0, 0x10000)
- *end character* - 1 is within range [0, 0x10000)
- *end line* is greater or equal to *start line*.
- *start line* is equal to *end line* then *end character* is greater than *start character*.

Note that the numbers specified in the directive syntax are 1-based numbers but the actual spans in the PDB are zero-based. Hence the -1 adjustments above.

The mapped spans of sequence points and the locations of diagnostics that `#line` directive applies to are calculated as follows.

Let d be the zero-based number of the unmapped line containing the `#line` directive. Let span $L = (\text{start: } (\text{start line} - 1, \text{start character} - 1), \text{end: } (\text{end line} - 1, \text{end character} - 1))$ be zero-based span specified by the directive.

Function M that maps a position (line, character) within the scope of the `#line` directive in the source file containing the `#line` directive to a mapped position (mapped line, mapped character) is defined as follows:

$M(l, c) =$

$l = d + 1 \Rightarrow (L.\text{start.line} + l - d - 1, L.\text{start.character} + \max(c - \text{character offset}, 0))$ $l > d + 1 \Rightarrow (L.\text{start.line} + l - d - 1, c)$

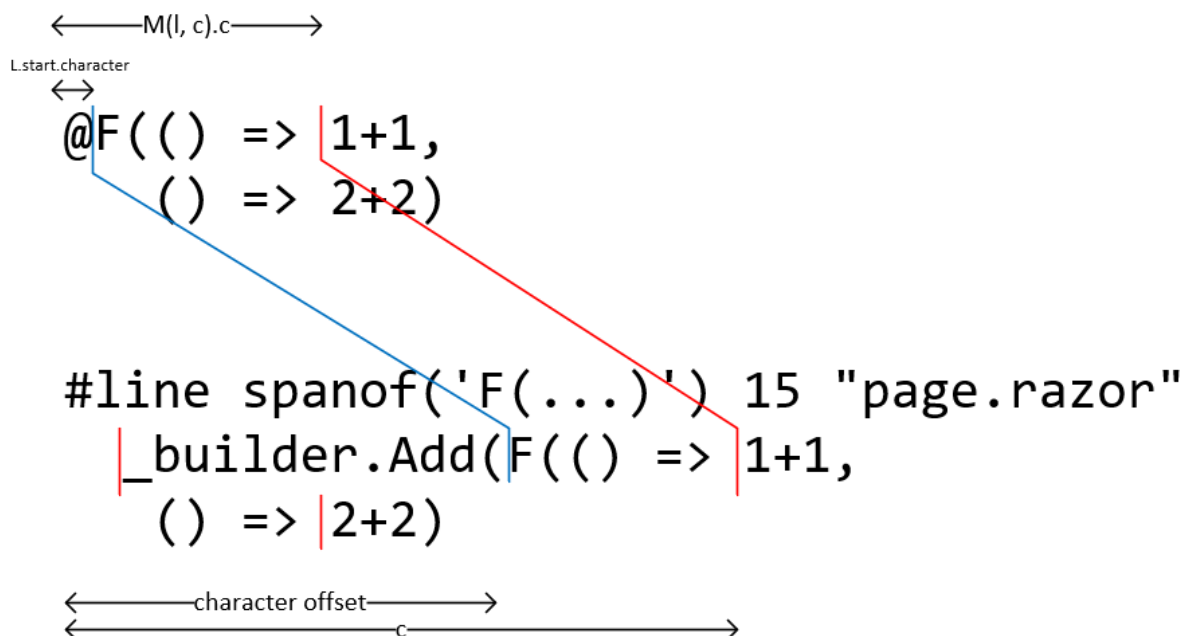
The syntax constructs that sequence points are associated with are determined by the compiler implementation and not covered by this specification. The compiler also decides for each sequence point its unmapped span. This span may partially or fully cover the associated syntax construct.

Once the unmapped spans are determined by the compiler the function M defined above is applied to their starting and ending positions, with the exception of the ending position of all sequence points within the scope of the `#line` directive whose unmapped location is at line $d + 1$ and character less than character offset. The end position of all these sequence points is $L.\text{end}$.

Example [5.i] demonstrates why it is necessary to provide the ability to specify the end position of the first sequence point span.

The above definition allows the generator of the unmapped source code to avoid intimate knowledge of which exact source constructs of the C# language produce sequence points. The mapped spans of the sequence points in the scope of the `#line` directive are derived from the relative position of the corresponding unmapped spans to the first unmapped span.

Specifying the *character offset* allows the generator to insert any single-line prefix on the first line. This prefix is generated code that is not present in the mapped file. Such inserted prefix affects the value of the first unmapped sequence point span. Therefore the starting character of subsequent sequence point spans need to be offset by the length of the prefix (*character offset*). See example [2].



Examples

For clarity the examples use `spanof('...')` and `lineof('...')` pseudo-syntax to express the mapped span start position and line number, respectively, of the specified code snippet.

1. First and subsequent spans

Consider the following code with unmapped zero-based line numbers listed on the right:

```
#line 1 10 1 15 "a"    // 3
  A();B(              // 4
);C();                // 5
  D();                // 6
```

$d = 3 \quad L = (0, 9)..(0, 14)$

There are 4 sequence point spans the directive applies to with following unmapped and mapped spans: $(4, 2)..(4, 5) \Rightarrow (0, 9)..(0, 14)$ $(4, 6)..(5, 1) \Rightarrow (0, 15)..(1, 1)$ $(5, 2)..(5, 5) \Rightarrow (1, 2)..(1, 5)$ $(6, 4)..(6, 7) \Rightarrow (2, 4)..(2, 7)$

2. Character offset

Razor generates `_builder.Add(` prefix of length 15 (including two leading spaces).

Razor:

```
@page "/"
@F( ) => 1+1,
    ( ) => 2+2
)
```

Generated C#:

```
#line hidden
void Render()
{
#line spanof('F(...)') 15 "page.razor" // 4
  _builder.Add(F( ) => 1+1,           // 5
    ( ) => 2+2                        // 6
  ));                                // 7
#line hidden
}
);
}
```

$d = 4 \quad L = (1, 1)..(3, 0)$ *character offset* = 15

Spans:

- `_builder.Add(F(...));` \Rightarrow `F(...)` : $(5, 2)..(7, 2) \Rightarrow (1, 1)..(3, 0)$
- `1+1` \Rightarrow `1+1` : $(5, 23)..(5, 25) \Rightarrow (1, 9)..(1, 11)$
- `2+2` \Rightarrow `2+2` : $(6, 7)..(6, 9) \Rightarrow (2, 7)..(2, 9)$

3. Razor: Single-line span

Razor:

```
@page "/"
Time: @DateTime.Now
```

Generated C#:

```
#line hidden
void Render()
{
    _builder.Add("Time:");
#line spanof('DateTime.Now') 15 "page.razor"
    _builder.Add(DateTime.Now);
#line hidden
};
}
```

4. Razor: Multi-line span

Razor:

```
@page "/"
@JsonToHtml(@"
{
    ""key1"": "value1",
    ""key2"": "value2"
}")
```

Generated C#:

```
#line hidden
void Render()
{
    _builder.Add("Time:");
#line spanof('JsonToHtml(@"..."') 15 "page.razor"
    _builder.Add(JsonToHtml(@"
{
    ""key1"": "value1",
    ""key2"": "value2"
}"));
#line hidden
}
};
}
```

5. Razor: block constructs

i. block containing expressions

In this example, the mapped span of the first sequence point that is associated with the IL instruction that is emitted for the `_builder.Add(Html.Helper(() =>` statement needs to cover the whole expression of `Html.Helper(...)` in the generated file `a.razor`. This is achieved by application of rule [1] to the end position of the sequence point.

```
@Html.Helper(() =>
{
    <p>Hello World</p>
    @DateTime.Now
})
```

```
#line spanof('Html.Helper(() => { ... }')) 13 "a.razor"
_builder.Add(Html.Helper(() =>
#line lineof('{}') "a.razor"
{
#line spanof('DateTime.Now') 13 "a.razor"
_builder.Add(DateTime.Now);
#line lineof('{}') "a.razor"
}
#line hidden
)
```

ii. block containing statements

Uses existing `#line line file` form since

a) Razor does not add any prefix, b) `{` is not present in the generated file and there can't be a sequence point placed on it, therefore the span of the first unmapped sequence point is unknown to Razor.

The starting character of `Console` in the generated file must be aligned with the Razor file.

```
@{Console.WriteLine(1);Console.WriteLine(2);}
```

```
#line lineof('@{}') "a.razor"
    Console.WriteLine(1);Console.WriteLine(2);
#line hidden
```

iii. block containing top-level code (@code, @functions)

Uses existing `#line line file` form since

a) Razor does not add any prefix, b) `{` is not present in the generated file and there can't be a sequence point placed on it, therefore the span of the first unmapped sequence point is unknown to Razor.

The starting character of `[Parameter]` in the generated file must be aligned with the Razor file.

```
@code {
    [Parameter]
    public int IncrementAmount { get; set; }
}
```

```
#line lineof('[{}') "a.razor"
    [Parameter]
    public int IncrementAmount { get; set; }
#line hidden
```

6. Razor: `@for`, `@foreach`, `@while`, `@do`, `@if`, `@switch`, `@using`, `@try`, `@lock`

Uses existing `#line line file` form since a) Razor does not add any prefix. b) the span of the first unmapped sequence point may not be known to Razor (or shouldn't need to know).

The starting character of the keyword in the generated file must be aligned with the Razor file.

```
@for (var i = 0; i < 10; i++)  
{  
}  
@if (condition)  
{  
}  
else  
{  
}
```

```
#line lineof('for') "a.razor"  
for (var i = 0; i < 10; i++)  
{  
}  
#line lineof('if') "a.razor"  
if (condition)  
{  
}  
else  
{  
}  
#line hidden
```


Generic Attributes

12/28/2021 • 2 minutes to read • [Edit Online](#)

Summary

When generics were introduced in C# 2.0, attribute classes were not allowed to participate. We can make the language more composable by removing (rather, loosening) this restriction. The .NET Core runtime has added support for generic attributes. Now, all that's missing is support for generic attributes in the compiler.

Motivation

Currently attribute authors can take a `System.Type` as a parameter and have users pass a `typeof` expression to provide the attribute with types that it needs. However, outside of analyzers, there's no way for an attribute author to constrain what types are allowed to be passed to an attribute via `typeof`. If attributes could be generic, then attribute authors could use the existing system of type parameter constraints to express the requirements for the types they take as input.

Detailed design

The following section is amended: <https://github.com/dotnet/csharplang/blob/main/spec/classes.md#base-classes>

The direct base class of a class type must not be any of the following types: `System.Array`, `System.Delegate`, `System.MulticastDelegate`, `System.Enum`, or `System.ValueType`. Furthermore, a generic class declaration cannot use `System.Attribute` as a direct or indirect base class.

One important note is that the following section of the spec is *unaffected* when referencing the point of usage of an attribute, i.e. within an attribute list: <https://github.com/dotnet/csharplang/blob/main/spec/types.md#type-parameters>

A type parameter cannot be used anywhere within an attribute.

This means that when a generic attribute is used, its construction needs to be fully "closed", i.e. not containing any type parameters, which means the following is still disallowed:

```
using System;
using System.Collections.Generic;

public class Attr<T1> : Attribute { }

public class Program<T2>
{
    [Attr<T2>] // error
    [Attr<List<T2>>] // error
    void M() { }
}
```

When a generic attribute is used in an attribute list, its type arguments have the same restrictions that `typeof` has on its argument. For example, `[Attr<dynamic>]` is an error. This is because "attribute-dependent" types like `dynamic`, `List<string?>`, `nint`, and so on can't be fully represented in the final IL for an attribute type argument, because there isn't a symbol to "attach" the `DynamicAttribute` or other well-known attribute to.

Drawbacks

Removing the restriction, reasoning out the implications, and adding the appropriate tests is work.

Alternatives

Attribute authors who want users to be able to discover the requirements for the types they provide to attributes need to write analyzers and guide their users to use those analyzers in their builds.

Unresolved questions

- [x] What does `AllowMultiple = false` mean on a generic attribute? If we have `[Attr<string>]` and `[Attr<object>]` both used on a symbol, does that mean "multiple" of the attribute are in use?
 - For now we are inclined to take the more restrictive route here and consider the attribute class's original definition when deciding whether multiple of it have been applied. In other words, `[Attr<string>]` and `[Attr<object>]` applied together is incompatible with `AllowMultiple = false`.

Design meetings

- <https://github.com/dotnet/csharplang/blob/main/meetings/2017/LDM-2017-02-21.md#generic-attributes>
 - At the time there was a concern that we would have to gate the feature on whether the target runtime supports it. (However, we now only support C# 10 on .NET 6. It would be nice to for the implementation to be aware of what minimum target framework supports the feature, but seems less essential today.)

Improved Definite Assignment Analysis

12/28/2021 • 13 minutes to read • [Edit Online](#)

Summary

[Definite assignment analysis](#) as specified has a few gaps which have caused users inconvenience. In particular, scenarios involving comparison to boolean constants, conditional-access, and null coalescing.

Related discussions and issues

csharp-lang discussion of this proposal: <https://github.com/dotnet/csharp-lang/discussions/4240>

Probably a dozen or so user reports can be found via this or similar queries (i.e. search for "definite assignment" instead of "CS0165", or search in csharp-lang). <https://github.com/dotnet/roslyn/issues?q=is%3Aclosed+is%3Aissue+label%3A%22Resolution-By+Design%22+cs0165>

I have included related issues in the scenarios below to give a sense of the relative impact of each scenario.

Scenarios

As a point of reference, let's start with a well-known "happy case" that does work in definite assignment and in nullable.

```
#nullable enable

C c = new C();
if (c != null && c.M(out object obj0))
{
    obj0.ToString(); // ok
}

public class C
{
    public bool M(out object obj)
    {
        obj = new object();
        return true;
    }
}
```

Comparison to bool constant

- <https://github.com/dotnet/csharp-lang/discussions/801>
- <https://github.com/dotnet/roslyn/issues/45582>
 - Links to 4 other issues where people were affected by this.

```

if ((c != null && c.M(out object obj1)) == true)
{
    obj1.ToString(); // undesired error
}

if ((c != null && c.M(out object obj2)) is true)
{
    obj2.ToString(); // undesired error
}

```

Comparison between a conditional access and a constant value

- <https://github.com/dotnet/roslyn/issues/33559>
- <https://github.com/dotnet/csharp-lang/discussions/4214>
- <https://github.com/dotnet/csharp-lang/issues/3659>
- <https://github.com/dotnet/csharp-lang/issues/3485>
- <https://github.com/dotnet/csharp-lang/issues/3659>

This scenario is probably the biggest one. We do support this in nullable but not in definite assignment.

```

if (c?.M(out object obj3) == true)
{
    obj3.ToString(); // undesired error
}

```

Conditional access coalesced to a bool constant

- <https://github.com/dotnet/csharp-lang/discussions/916>
- <https://github.com/dotnet/csharp-lang/issues/3365>

This scenario is very similar to the previous one. This is also supported in nullable but not in definite assignment.

```

if (c?.M(out object obj4) ?? false)
{
    obj4.ToString(); // undesired error
}

```

Conditional expressions where one arm is a bool constant

- <https://github.com/dotnet/roslyn/issues/4272>

It's worth pointing out that we already have special behavior for when the condition expression is constant (i.e. `true ? a : b`). We just unconditionally visit the arm indicated by the constant condition and ignore the other arm.

Also note that we haven't handled this scenario in nullable.

```

if (c != null ? c.M(out object obj4) : false)
{
    obj4.ToString(); // undesired error
}

```

Specification

?. (null-conditional operator) expressions

We introduce a new section **?. (null-conditional operator) expressions**. See the [null-conditional operator specification](#) and [definite assignment rules](#) for context.

As in the definite assignment rules linked above, we refer to a given initially unassigned variable as v .

We introduce the concept of "directly contains". An expression E is said to "directly contain" a subexpression E_1 if it is not subject to a [user-defined conversion](#) whose parameter is not of a non-nullable value type, and one of the following conditions holds:

- E is E_1 . For example, `a?.b()` directly contains the expression `a?.b()`.
- If E is a parenthesized expression `(E2)`, and E_2 directly contains E_1 .
- If E is a null-forgiving operator expression `E2!`, and E_2 directly contains E_1 .
- If E is a cast expression `(T)E2`, and the cast does not subject E_2 to a non-lifted user-defined conversion whose parameter is not of a non-nullable value type, and E_2 directly contains E_1 .

For an expression E of the form `primary_expression null_conditional_operations`, let E_0 be the expression obtained by textually removing the leading `?` from each of the *null_conditional_operations* of E that have one, as in the linked specification above.

In subsequent sections we will refer to E_0 as the *non-conditional counterpart* to the null-conditional expression. Note that some expressions in subsequent sections are subject to additional rules that only apply when one of the operands directly contains a null-conditional expression.

- The definite assignment state of v at any point within E is the same as the definite assignment state at the corresponding point within E_0 .
- The definite assignment state of v after E is the same as the definite assignment state of v after *primary_expression*.

Remarks

We use the concept of "directly contains" to allow us to skip over relatively simple "wrapper" expressions when analyzing conditional accesses that are compared to other values. For example, `((a?.b(out x))!) == true` is expected to result in the same flow state as `a?.b == true` in general.

We also want to allow analysis to function in the presence of a number of possible conversions on a conditional access. Propagating out "state when not null" is not possible when the conversion is user-defined, though, since we can't count on user-defined conversions to honor the constraint that the output is non-null only if the input is non-null. The only exception to this is when the user-defined conversion's input is a non-nullable value type. For example:

```
public struct S1 { }
public struct S2 { public static implicit operator S2?(S1 s1) => null; }
```

This also includes lifted conversions like the following:

```

string x;

S1? s1 = null;
_ = s1?.M1(x = "a") ?? s1.Value.M2(x = "a");

x.ToString(); // ok

public struct S1
{
    public S1 M1(object obj) => this;
    public S2 M2(object obj) => new S2();
}
public struct S2
{
    public static implicit operator S2(S1 s1) => null;
}

```

When we consider whether a variable is assigned at a given point within a null-conditional expression, we simply assume that any preceding null-conditional operations within the same null-conditional expression succeeded.

For example, given a conditional expression `a?.b(out x)?.c(x)`, the non-conditional counterpart is `a.b(out x).c(x)`. If we want to know the definite assignment state of `x` before `?.c(x)`, for example, then we perform a "hypothetical" analysis of `a.b(out x)` and use the resulting state as an input to `?.c(x)`.

Boolean constant expressions

We introduce a new section "Boolean constant expressions":

For an expression *expr* where *expr* is a constant expression with a bool value:

- The definite assignment state of *v* after *expr* is determined by:
 - If *expr* is a constant expression with value *true*, and the state of *v* before *expr* is "not definitely assigned", then the state of *v* after *expr* is "definitely assigned when false".
 - If *expr* is a constant expression with value *false*, and the state of *v* before *expr* is "not definitely assigned", then the state of *v* after *expr* is "definitely assigned when true".

Remarks

We assume that if an expression has a constant value bool `false`, for example, it's impossible to reach any branch that requires the expression to return `true`. Therefore variables are assumed to be definitely assigned in such branches. This ends up combining nicely with the spec changes for expressions like `??` and `?:` and enabling a lot of useful scenarios.

It's also worth noting that we never expect to be in a conditional state *before* visiting a constant expression. That's why we do not account for scenarios such as "*expr* is a constant expression with value *true*, and the state of *v* before *expr* is "definitely assigned when true".

?? (null-coalescing expressions) augment

We augment the section [?? \(null coalescing\) expressions](#) as follows:

For an expression *expr* of the form `expr_first ?? expr_second`:

- ...
- The definite assignment state of *v* after *expr* is determined by:
 - ...
 - If *expr_first* directly contains a null-conditional expression *E*, and *v* is definitely assigned after the non-conditional counterpart *E₀*, then the definite assignment state of *v* after *expr* is the same as the definite assignment state of *v* after *expr_second*.

Remarks

The above rule formalizes that for an expression like `a?.M(out x) ?? (x = false)`, either the `a?.M(out x)` was fully evaluated and produced a non-null value, in which case `x` was assigned, or the `x = false` was evaluated, in which case `x` was also assigned. Therefore `x` is always assigned after this expression.

This also handles the `dict?.TryGetValue(key, out var value) ?? false` scenario, by observing that `v` is definitely assigned after `dict.TryGetValue(key, out var value)`, and `v` is "definitely assigned when true" after `false`, and concluding that `v` must be "definitely assigned when true".

The more general formulation also allows us to handle some more unusual scenarios, such as:

- `if (x?.M(out y) ?? (b && z.M(out y))) y.ToString();`
- `if (x?.M(out y) ?? z?.M(out y) ?? false) y.ToString();`

?: (conditional) expressions

We augment the section [?: \(conditional\) expressions](#) as follows:

For an expression *expr* of the form `expr_cond ? expr_true : expr_false`:

- ...
- The definite assignment state of *v* after *expr* is determined by:
 - ...
 - If the state of *v* after *expr_true* is "definitely assigned when true", and the state of *v* after *expr_false* is "definitely assigned when true", then the state of *v* after *expr* is "definitely assigned when true".
 - If the state of *v* after *expr_true* is "definitely assigned when false", and the state of *v* after *expr_false* is "definitely assigned when false", then the state of *v* after *expr* is "definitely assigned when false".

Remarks

This makes it so when both arms of a conditional expression result in a conditional state, we join the corresponding conditional states and propagate it out instead of unsplitting the state and allowing the final state to be non-conditional. This enables scenarios like the following:

```
bool b = true;
object x = null;
int y;
if (b ? x != null && Set(out y) : x != null && Set(out y))
{
    y.ToString();
}

bool Set(out int x) { x = 0; return true; }
```

This is an admittedly niche scenario, that compiles without error in the native compiler, but was broken in Roslyn in order to match the specification at the time. See [internal issue](#).

==/!= (relational equality operator) expressions

We introduce a new section [==/!= \(relational equality operator\) expressions](#).

The [general rules for expressions with embedded expressions](#) apply, except for the scenarios described below.

For an expression *expr* of the form `expr_first == expr_second`, where `==` is a [predefined comparison operator](#) or a [lifted operator](#), the definite assignment state of *v* after *expr* is determined by:

- If *expr_first* directly contains a null-conditional expression *E* and *expr_second* is a constant expression with value *null*, and the state of *v* after the non-conditional counterpart *E₀* is "definitely assigned", then the state of *v* after *expr* is "definitely assigned when false".
- If *expr_first* directly contains a null-conditional expression *E* and *expr_second* is an expression of a non-

nullable value type, or a constant expression with a non-null value, and the state of v after the non-conditional counterpart E_0 is "definitely assigned", then the state of v after $expr$ is "definitely assigned when true".

- If $expr_first$ is of type *boolean*, and $expr_second$ is a constant expression with value *true*, then the definite assignment state after $expr$ is the same as the definite assignment state after $expr_first$.
- If $expr_first$ is of type *boolean*, and $expr_second$ is a constant expression with value *false*, then the definite assignment state after $expr$ is the same as the definite assignment state of v after the logical negation expression `!expr_first`.

For an expression $expr$ of the form `expr_first != expr_second`, where `!=` is a [predefined comparison operator](#) or a [lifted operator](#), the definite assignment state of v after $expr$ is determined by:

- If $expr_first$ directly contains a null-conditional expression E and $expr_second$ is a constant expression with value *null*, and the state of v after the non-conditional counterpart E_0 is "definitely assigned", then the state of v after $expr$ is "definitely assigned when true".
- If $expr_first$ directly contains a null-conditional expression E and $expr_second$ is an expression of a non-null value type, or a constant expression with a non-null value, and the state of v after the non-conditional counterpart E_0 is "definitely assigned", then the state of v after $expr$ is "definitely assigned when false".
- If $expr_first$ is of type *boolean*, and $expr_second$ is a constant expression with value *true*, then the definite assignment state after $expr$ is the same as the definite assignment state of v after the logical negation expression `!expr_first`.
- If $expr_first$ is of type *boolean*, and $expr_second$ is a constant expression with value *false*, then the definite assignment state after $expr$ is the same as the definite assignment state after $expr_first$.

All of the above rules in this section are commutative, meaning that if a rule applies when evaluated in the form `expr_second op expr_first`, it also applies in the form `expr_first op expr_second`.

Remarks

The general idea expressed by these rules is:

- if a conditional access is compared to `null`, then we know the operations definitely occurred if the result of the comparison is `false`
- if a conditional access is compared to a non-null value type or a non-null constant, then we know the operations definitely occurred if the result of the comparison is `true`.
- since we can't trust user-defined operators to provide reliable answers where initialization safety is concerned, the new rules only apply when a predefined `==` / `!=` operator is in use.

We may eventually want to refine these rules to thread through conditional state which is present at the end of a member access or call. Such scenarios don't really happen in definite assignment, but they do happen in nullable in the presence of `[NotNullWhen(true)]` and similar attributes. This would require special handling for `bool` constants in addition to just handling for `null` /non-null constants.

Some consequences of these rules:

- `if (a?.b(out var x) == true)) x() else x();` will error in the 'else' branch
- `if (a?.b(out var x) == 42)) x() else x();` will error in the 'else' branch
- `if (a?.b(out var x) == false)) x() else x();` will error in the 'else' branch
- `if (a?.b(out var x) == null)) x() else x();` will error in the 'then' branch
- `if (a?.b(out var x) != true)) x() else x();` will error in the 'then' branch
- `if (a?.b(out var x) != 42)) x() else x();` will error in the 'then' branch
- `if (a?.b(out var x) != false)) x() else x();` will error in the 'then' branch
- `if (a?.b(out var x) != null)) x() else x();` will error in the 'else' branch

is operator and is pattern expressions

We introduce a new section `is operator and is pattern expressions`.

For an expression *expr* of the form `E is T`, where *T* is any type or pattern

- The definite assignment state of *v* before *E* is the same as the definite assignment state of *v* before *expr*.
- The definite assignment state of *v* after *expr* is determined by:
 - If *E* directly contains a null-conditional expression, and the state of *v* after the non-conditional counterpart *E₀* is "definitely assigned", and `T` is any type or a pattern that does not match a `null` input, then the state of *v* after *expr* is "definitely assigned when true".
 - If *E* directly contains a null-conditional expression, and the state of *v* after the non-conditional counterpart *E₀* is "definitely assigned", and `T` is a pattern that matches a `null` input, then the state of *v* after *expr* is "definitely assigned when false".
 - If *E* is of type boolean and `T` is a pattern which only matches a `true` input, then the definite assignment state of *v* after *expr* is the same as the definite assignment state of *v* after *E*.
 - If *E* is of type boolean and `T` is a pattern which only matches a `false` input, then the definite assignment state of *v* after *expr* is the same as the definite assignment state of *v* after the logical negation expression `!expr`.
 - Otherwise, if the definite assignment state of *v* after *E* is "definitely assigned", then the definite assignment state of *v* after *expr* is "definitely assigned".

Remarks

This section is meant to address similar scenarios as in the `== / !=` section above. This specification does not address recursive patterns, e.g. `(a?.b(out x), c?.d(out y)) is (object, object)`. Such support may come later if time permits.

Additional scenarios

This specification doesn't currently address scenarios involving pattern switch expressions and switch statements. For example:

```
_ = c?.M(out object obj4) switch
{
    not null => obj4.ToString() // undesired error
};
```

It seems like support for this could come later if time permits.

There have been several categories of bugs filed for nullable which require we essentially increase the sophistication of pattern analysis. It is likely that any ruling we make which improves definite assignment would also be carried over to nullable.

<https://github.com/dotnet/roslyn/issues/49353>

<https://github.com/dotnet/roslyn/issues/46819>

<https://github.com/dotnet/roslyn/issues/44127>

Drawbacks

It feels odd to have the analysis "reach down" and have special recognition of conditional accesses, when typically flow analysis state is supposed to propagate upward. We are concerned about how a solution like this could intersect painfully with possible future language features that do null checks.

Alternatives

Two alternatives to this proposal:

1. Introduce "state when null" and "state when not null" to the language and compiler. This has been judged to be too much effort for the scenarios we are trying to solve, but that we could potentially implement the above proposal and then move to a "state when null/not null" model later on without breaking people.
2. Do nothing.

Unresolved questions

There are impacts on switch expressions that should be specified:

<https://github.com/dotnet/csharplang/discussions/4240#discussioncomment-343395>

Design meetings

<https://github.com/dotnet/csharplang/discussions/4243>

AsyncMethodBuilder override

12/28/2021 • 8 minutes to read • [Edit Online](#)

Summary

Allow per-method override of the async method builder to use. For some async methods we want to customize the invocation of `Builder.Create()` to use a different *builder type*.

```
[AsyncMethodBuilderAttribute(typeof(PoolingAsyncValueTaskMethodBuilder<>))] // new usage of
AsyncMethodBuilderAttribute type
static async ValueTask<int> ExampleAsync() { ... }
```

Motivation

Today, async method builders are tied to a given type used as a return type of an async method. For example, any method that's declared as `async Task` uses `AsyncTaskMethodBuilder`, and any method that's declared as `async ValueTask<T>` uses `AsyncValueTaskMethodBuilder<T>`. This is due to the `[AsyncMethodBuilder(Type)]` attribute on the type used as a return type, e.g. `ValueTask<T>` is attributed as `[AsyncMethodBuilder(typeof(AsyncValueTaskMethodBuilder<>))]`. This addresses the majority common case, but it leaves a few notable holes for advanced scenarios.

In .NET 5, an experimental feature was shipped that provides two modes in which `AsyncValueTaskMethodBuilder` and `AsyncValueTaskMethodBuilder<T>` operate. The on-by-default mode is the same as has been there since the functionality was introduced: when the state machine needs to be lifted to the heap, an object is allocated to store the state, and the async method returns a `ValueTask<T>` backed by a `Task<T>`. However, if an environment variable is set, all builders in the process switch to a mode where, instead, the `ValueTask<T>` instances are backed by reusable `IValueTaskSource<T>` implementations that are pooled. Each async method has its own pool with a fixed maximum number of instances allowed to be pooled, and as long as no more than that number are ever returned to the pool to be pooled at the same time, `async ValueTask<T>` methods effectively become free of any GC allocation overhead.

There are several problems with this experimental mode, however, which is both why a) it's off by default and b) we're likely to remove it in a future release unless very compelling new information emerges (<https://github.com/dotnet/runtime/issues/13633>).

- It introduces a behavioral difference for consumers of the returned `ValueTask<T>` if that `ValueTask` isn't being consumed according to spec. When it's backed by a `Task`, you can do with the `ValueTask` things you can do with a `Task`, like await it multiple times, await it concurrently, block waiting for it to complete, etc. But when it's backed by an arbitrary `IValueTaskSource`, such operations are prohibited, and automatically switching from the former to the latter can lead to bugs. With the switch at the process level and affecting all `async ValueTask` methods in the process, whether you control them or not, it's too big a hammer.
- It's not necessarily a performance win, and could represent a regression in some situations. The implementation is trading the cost of pooling (accessing a pool isn't free) with the cost of GC, and in various situations the GC can win. Again, applying the pooling to all `async ValueTask` methods in the process rather than being selective about the ones it would most benefit is too big a hammer.
- It adds to the IL size of a trimmed application, even if the flag isn't set, and then to the resulting asm size. It's possible that can be worked around with improvements to the implementation to teach it that for a given deployment the environment variable will always be false, but as it stands today, every `async ValueTask` method saw for example an ~2K binary footprint increase in aot images due to this option, and, again, that

applies to all `async ValueTask` methods in the whole application closure.

- Different methods may benefit from differing levels of control, e.g. the size of the pool employed because of knowledge of the method and how it's used, but the same setting is applied to all uses of the builder. One could imagine working around that by having the builder code use reflection at runtime to look for some attribute, but that adds significant run-time expense, and likely on the startup path.

On top of all of these issues with the existing pooling, it's also the case that developers are prevented from writing their own customized builders for types they don't own. If, for example, a developer wants to implement their own pooling support, they also have to introduce a brand new task-like type, rather than just being able to use `{Value}Task{<T>}`, because the attribute specifying the builder is only specifiable on the type declaration of the return type.

We need a way to have an individual async method opt-in to a specific builder.

Detailed design

Using AsyncMethodBuilderAttribute on methods

In `dotnet/runtime`, add `AttributeTargets.Method` to the targets for `System.Runtime.CompilerServices.AsyncMethodBuilderAttribute`:

```
namespace System.Runtime.CompilerServices
{
    /// <summary>
    /// Indicates the type of the async method builder that should be used by a language compiler:
    /// - to build the return type of an async method that is attributed,
    /// - to build the attributed type when used as the return type of an async method.
    /// </summary>
    [AttributeUsage(AttributeTargets.Method | AttributeTargets.Class | AttributeTargets.Struct |
        AttributeTargets.Interface | AttributeTargets.Delegate | AttributeTargets.Enum, Inherited = false,
        AllowMultiple = false)]
    public sealed class AsyncMethodBuilderAttribute : Attribute
    {
        /// <summary>Initializes the <see cref="AsyncMethodBuilderAttribute"/>.</summary>
        /// <param name="builderType">The <see cref="Type"/> of the associated builder.</param>
        public AsyncMethodBuilderAttribute(Type builderType) => BuilderType = builderType;

        /// <summary>Gets the <see cref="Type"/> of the associated builder.</summary>
        public Type BuilderType { get; }
    }
}
```

This allows the attribute to be applied on methods or local functions or lambdas.

Example of usage on a method:

```
[AsyncMethodBuilder(typeof(PoolingAsyncValueTaskMethodBuilder<>))] // new usage, referring to some custom
builder type
static async ValueTask<int> ExampleAsync() { ... }
```

It is an error to apply the attribute multiple times on a given method.

It is an error to apply the attribute to a lambda with an implicit return type.

A developer who wants to use a specific custom builder for all of their methods can do so by putting the relevant attribute on each method.

Determining the builder type for an async method

When compiling an async method, the builder type is determined by:

1. using the builder type from the `AsyncMethodBuilder` attribute if one is present,
2. otherwise, falling back to the builder type determined by previous approach. (see [spec for task-like types](#)).

If an `AsyncMethodBuilder` attribute is present, we take the builder type specified by the attribute and construct it if necessary.

If the override type is an open generic type, take the single type argument of the async method's return type and substitute it into the override type.

If the override type is a bound generic type, then we produce an error.

If the async method's return type does not have a single type argument, then we produce an error.

We verify that the builder type is compatible with the return type of the async method:

1. look for the public `Create` method with no type parameters and no parameters on the constructed builder type.
It is an error if the method is not found. It is an error if the method returns a type other than the constructed builder type.
2. look for the public `Task` property.
It is an error if the property is not found.
3. consider the type of that `Task` property (a task-like type):
It is an error if the task-like type does not matches the return type of the async method.

Note that it is not necessary for the return type of the method to be a task-like type.

Execution

The builder type determined above is used as part of the existing async method design.

For example, today if a method is defined as:

```
public async ValueTask<T> ExampleAsync() { ... }
```

the compiler will generate code akin to:

```
[AsyncStateMachine(typeof(<ExampleAsync>d__29))]
[CompilerGenerated]
static ValueTask<int> ExampleAsync()
{
    <ExampleAsync>d__29 stateMachine;
    stateMachine.<>t__builder = AsyncValueTaskMethodBuilder<int>.Create();
    stateMachine.<>1__state = -1;
    stateMachine.<>t__builder.Start(ref stateMachine);
    return stateMachine.<>t__builder.Task;
}
```

With this change, if the developer wrote:

```
[AsyncMethodBuilder(typeof(PoolingAsyncValueTaskMethodBuilder<>))] // new usage, referring to some custom
builder type
static async ValueTask<int> ExampleAsync() { ... }
```

it would instead be compiled to:

```
[AsyncStateMachine(typeof(<ExampleAsync>d__29))]
[CompilerGenerated]
[AsyncMethodBuilder(typeof(PoolingAsyncValueTaskMethodBuilder<>))] // retained but not necessary anymore
static ValueTask<int> ExampleAsync()
{
    <ExampleAsync>d__29 stateMachine;
    stateMachine.<>t__builder = PoolingAsyncValueTaskMethodBuilder<int>.Create(); // <>t__builder now a
different type
    stateMachine.<>1__state = -1;
    stateMachine.<>t__builder.Start(ref stateMachine);
    return stateMachine.<>t__builder.Task;
}
```

Just those small additions enable:

- Anyone to write their own builder that can be applied to async methods that return `Task<T>` and `ValueTask<T>`
- As "anyone", the runtime to ship the experimental builder support as new public builder types that can be opted into on a method-by-method basis; the existing support would be removed from the existing builders. Methods (including some we care about in the core libraries) can then be attributed on a case-by-case basis to use the pooling support, without impacting any other unattributed methods.

and with minimal surface area changes or feature work in the compiler.

Note that we need the emitted code to allow a different type being returned from `Create` method:

```
AsyncPooledBuilder _builder = AsyncPooledBuilderWithSize4.Create();
```

Note that this mechanism to change the the builder type cannot be used when the synthesized entry-point for top-level statements is async. An explicit entry-point should be used instead.

Drawbacks

- The syntax for applying such an attribute to a method is verbose. The impact of this is lessened if a developer can apply it to multiple methods en mass, e.g. at the type or module level.

Alternatives

- Implement a different task-like type and expose that difference to consumers. `ValueTask` was made extensible via the `IValueTaskSource` interface to avoid that need, however.
- Address just the `ValueTask` pooling part of the issue by enabling the experiment as the on-by-default-and-only implementation. That doesn't address other aspects, such as configuring the pooling, or enabling someone else to provide their own builder.
- Earlier versions of this document allowed for scoped override of builder types.

Unresolved questions

1. **Replace or also create.** All of the examples in this proposal are about replacing a buildable task-like's builder. Should the feature be scoped to just that? Or should you be able to use this attribute on a method with a return type that doesn't already have a builder (e.g. some common interface)? That could impact overload resolution.
2. **Private Builders.** Should the compiler support non-public async method builders? This is not spec'd today, but experimentally we only support public ones. That makes some sense when the attribute is applied to a type to control what builder is used with that type, since anyone writing an async method with that type as

the return type would need access to the builder. However, with this new feature, when that attribute is applied to a method, it only impacts the implementation of that method, and thus could reasonably reference a non-public builder. Likely we will want to support library authors who have non-public ones they want to use.